UiO : Department of Mathematics
University of Oslo

Iterative Solvers for Diffusion Equations Arising in Models of Calcium Dynamics

Thesis for the Degree of Master of Science
in Computational Science and Engineering

Ingrid Elgsaas-Vada – September 2015

Abstract

Reaction-Diffusion equations arise in many models in biomedical computing. One of these areas are models for the calcium dynamic in the heart cells. As the terms in reaction-diffusion equations can be solved separately using operator splitting it is important to have efficient and accurate ways of solving these parts. The focus of this thesis is solving the diffusion equation, particularly diffusion equations arising in models of calcium dynamics. We study a simple test problem and present alternative ways of discretizing this problem. Our focus has been on solving this test problem using an implicit scheme and iterative solvers. In this thesis we present the theory behind these solvers. We have also implemented and tested these solvers using a serial implementation in MATLAB and compared their performance to an explicit scheme. Based on the results of our serial tests we have implemented two of our iterative solvers using parallel programming in C++. Finally we have attempted to find which of the discussed iterative solvers perform best for our diffusion equation using parallel implementation. It is shown that implicit schemes combined with iterative equation solvers can be an interesting option for solving the diffusion equation.

Acknowledgements

I would like to thank my supervisors Glenn Terje Lines, Aslak Tveito and Knut Mørken for providing me with such an interesting thesis topic. I am especially grateful to Glenn Lines for the help and discussions during my work on this thesis. He has always taken the time to answer my questions and discuss the problems and has provided invaluable guidance and insight. I am grateful to Simula Research Laboratory for the possibility to do my thesis within their research and at the premises. It has been a challenging and interesting journey.

I would also like to thank Xing Cai for his help with the parallel implementations. Both for helping me get started on modifying the parallel code and for being available to answer any and all questions I have had when working on the parallel programming.

I would like to thank Simen Tennøe for taking the time to read through this thesis and checking that everything made sense.

Finally, I would like to thank my friends for their understanding and support throughout my work on this thesis. I would also like to thank my family for always being there, and I am especially grateful to my brother Gisle for his unending patience and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many different areas of mathematical modelling and mathematical theory requires us to solve systems of linear equations. This is especially relevant in computational mathematics as we have been able to solve larger and larger systems as the computational powers of computers has increased. By using parallel programming we are able to solve even larger problems in a reasonable time frame. To solve our problems as fast as possible, we are interested in using efficient algorithms for solving the linear systems. In this thesis we work on the systems of linear equations that result when solving partial differential equations with implicit methods. We will focus on a particular problem, namely that of efficiently solving the diffusion part of a reaction-diffusion equation.

1.1 Purpose

There are two ways of solving time-dependant partial differential equations. Either through the use of an explicit scheme or an implicit scheme. An explicit scheme allows us to solve the problem directly. Using an implicit scheme results in a system of equations we have to solve. When the partial differential equation is a linear equation this system of equations will be system of linear equations. We can then use linear algebra to efficiently solve this system.

There are of course both advantages and disadvantages regardless of whether one uses an explicit or an implicit scheme. With an explicit scheme the advantage is that you get the solution directly. They are also usually less computationally heavy than the implicit schemes. The disadvantage is that such schemes are generally subject to very strict stability conditions, which can result in, for instance, requiring very small time steps. And there are cases, like elliptical problems, where we can not use explicit schemes at all. The Explicit Euler scheme is an example of an explicit scheme.

In this thesis we will restrict ourselves to study a problem which yields

a linear system of equations when we use an implicit scheme. When using an implicit scheme to solve a partial differential equation we have to solve a system of linear equations which means that we can take advantage of mathematical theory for solving the matrix-vector equations,

$$\boldsymbol{Ax} = \boldsymbol{b}.$$

The simplest and most computationally expensive method is using Gaussian Elimination. This enables us to calculate the matrix-vector product $\boldsymbol{Ax}$ directly, but this operation is both time and memory consuming when performed on a computer. We are therefore interested in studying iterative methods and see if these methods can give us better results. These methods operate by instead of directly solving the matrix-vector product, they iterate to find an approximate solution, where each iteration brings us a little bit closer to the actual solution. The drawback of this is that we have to iterate, perhaps iterate a lot, to get the solution, while the explicit method gives us the solution in one step. We are going to study two types of iterative methods. The classical iterative methods, such as Jacobi's method, are relatively simple methods which do not need a lot of computations for each iteration, but might require quite a lot of iterations to converge. The other type is more sophisticated solvers like the Conjugate-Gradient method. These methods have heavier calculations in each iteration, but usually converges within a very small number of iterations. It is common to use either an explicit method like the Explicit Euler scheme, because of its simplicity, or a very complex scheme like the Conjugate-Gradient method, because of its quick convergence, to solve partial differential equations which yields linear systems. In this thesis we want to explore whether the classical iterative methods could be a useful alternative. As we are interested in solving very large problems, we are particularly interested in testing how these methods perform when using parallel programming.

Reaction-diffusion equations arise in many models in biomedical computing. In this thesis we will focus on a particular model of calcium dynamics in heart cells. We will explore how we can use theories from numerical linear algebra to effectively solve the diffusion part of these equations by studying a simplified test problem.

1.2 Structure

The structure of this thesis is as follows.

Chapter 2 explains the real life problem we ultimately are interested in solving. In this thesis we will restrict ourselves to look at efficiently solving the diffusion part of this model, but it is useful to have an overview of the mathematical model of the real life problem.

The test problem is presented and we discuss ways of discretizing this problem both with respect to time and space in Chapter 3.

Chapter 4 introduces the mathematical theory for the iterative methods we will test and discusses what the methods require to guarantee convergence. This last point is an important factor when determining what kind of problems these methods will be useful for.

Our test problem has been implemented in Matlab and we present various results from our serial tests and discuss how each method performs in Chapter 5.

As we also are interested in looking at the performance of our iterative methods when using parallel implementation, we have performed some testing on our problem using parallelization. We start by discussing the additional challenges that arise when we have to take communication between processes into account. This is presented in Chapter 6. In addition we have implemented some of the iterative methods in C++ so we can compare the actual performance with the theoretical performance. Which methods we chose to implement in parallel was based on the results from our serial tests and the expected communication requirements. Chapter 7 gives the results from our tests.

Chapter 8 gives the conclusion of this thesis and present some points which would be interesting to study further.

Chapter 2

Reaction-Diffusion Equations and Calcium Dynamics of the Heart

2.1 Calcium Dynamics

The calcium dynamics in heart cells are important for the heart to function. Releasing calcium into the cells so the intracellular Ca^{2+} concentration is increased is what causes the heart cells to contract. As this is the process that causes the heart to move blood around to the rest of the body it is very important that this functions properly. Calcium is released into the cells from many different release units. These releases are synchronous and stable in healthy heart cells and cause an almost instant and very large increase in the intracellular calcium concentration.

It is interesting to study mathematical models of this process because it is so central to how the heart functions. One problem with the models for these calcium dynamics is that they require immense amounts of computing power to solve, especially if we want to solve them on as small as a scale of a few nanometres. In the article Chai et al. (2013) the authors are working towards a simulation of these subcellular calcium dynamics on a nanometre scale. They are able to simulate these dynamics on a three nanometre resolution by using Tianhe-2 which is the most powerful supercomputer in the world.

2.2 The Model

Chai et al. (2013) used the model

$$\begin{aligned}
\frac{\partial c}{\partial t} &= \mathrm{D}_{\mathrm{Ca}}^{\mathrm{cyt}}\nabla^2 c + R_{\mathrm{SR}}(c, c^{\mathrm{sr}}) - \sum_i R_i(c, c^{B_i}),\\
\frac{\partial c^{\mathrm{sr}}}{\partial t} &= \mathrm{D}_{\mathrm{Ca}}^{\mathrm{sr}}\nabla^2 c^{\mathrm{sr}} - \frac{R_{\mathrm{SR}}(c, c^{\mathrm{sr}})}{\gamma} - R_{\mathrm{CSQN}}(c^{\mathrm{sr}}, c^{B_{\mathrm{CSQN}}}),\\
\frac{\partial c^{B_{\mathrm{ATP}}}}{\partial t} &= \mathrm{D}_{\mathrm{ATP}}^{\mathrm{cyt}}\nabla^2 c^{B_{\mathrm{ATP}}} + R_{\mathrm{ATP}}(c, c^{B_{\mathrm{ATP}}}),\\
\frac{\partial c^{B_{\mathrm{CMDN}}}}{\partial t} &= \mathrm{D}_{\mathrm{CMDN}}^{\mathrm{cyt}}\nabla^2 c^{B_{\mathrm{CMDN}}} + R_{\mathrm{CMDN}}(c, c^{B_{\mathrm{CMDN}}}),\\
\frac{\partial c^{B_{\mathrm{Fluo}}}}{\partial t} &= \mathrm{D}_{\mathrm{Fluo}}^{\mathrm{cyt}}\nabla^2 c^{B_{\mathrm{Fluo}}} + R_{\mathrm{Fluo}}(c, c^{B_{\mathrm{Fluo}}}),\\
\frac{\mathrm{d}c^{B_{\mathrm{TRPN}}}}{\mathrm{d}t} &= R_{\mathrm{TRPN}}(c, c^{B_{\mathrm{TRPN}}}),\\
\frac{\mathrm{d}c^{B_{\mathrm{CSQN}}}}{\mathrm{d}t} &= R_{\mathrm{CSQN}}(c^{\mathrm{sr}}, c^{B_{\mathrm{CSQN}}}),
\end{aligned}$$

This model consists of five reaction-diffusion equations and two ordinary differential equations. The seven primary unknowns are the Ca^2+ concentrations (c, c^{sr}, $c^{B_{\mathrm{ATP}}}$, $c^{B_{\mathrm{CMDN}}}$, $c^{B_{\mathrm{Fluo}}}$, $c^{B_{\mathrm{TRPN}}}$ and $c^{B_{\mathrm{CSQN}}}$), where c and c^{sr} are calcium concentrations while the other five are calcium buffers. The five D constants are the diffusion properties of the reaction-diffusion equations and the γ constant is the volume fraction of the sarcoplasmic reticulum (SR). The $R(\cdot,\cdot)$ are the reaction terms which can all be expressed by

$$R_i(c, c^{B_i}) = \mathrm{k}_{\mathrm{on}}^i c(\mathrm{B}_{\mathrm{tot}}^i - c^{B_i}) - \mathrm{k}_{\mathrm{off}}^i c^{B_i},$$

where $\mathrm{k}_{\mathrm{on}}^i$, $\mathrm{B}_{\mathrm{tot}}^i$ and $\mathrm{k}_{\mathrm{off}}^i$ are known constants. The only reaction term which can not be expressed this way is $R_{\mathrm{SR}}(c, c^{\mathrm{sr}})$ which instead can be expressed as

$$R_{\mathrm{SR}}(c, c^{\mathrm{sr}}) = R_{\mathrm{RyR}}(c, c^{\mathrm{sr}}) - R_{\mathrm{serca}}(c, c^{\mathrm{sr}}).$$

where

$$R_{\mathrm{RyR}}(c, c^{\mathrm{sr}}) = P_o(c)k(c^{\mathrm{sr}} - c),$$

and

$$R_{\mathrm{serca}}(c, c^{\mathrm{sr}}) = \frac{a_1 c \cdot c + a_2 c^{\mathrm{sr}} \cdot c^{\mathrm{sr}}}{a_3 c \cdot c + a_4 c^{\mathrm{sr}} \cdot c^{\mathrm{sr}} + a_5},$$

The variable $P_o(c)$ in $R_{\mathrm{RyR}}$is a binary variable. This model is described in more detail in Chai et al. (2013)

This model uses no-flow boundary conditions on the boundary of the three-dimensional computational domain.

2.3 Operator Splitting

It is common to solve problems like reaction-diffusion equations by using operator splitting. This allows us to separately solve the two parts of the equation instead of having to solve everything at once. This is an advantage as the reaction and diffusion parts of the equation usually can be solved very effectively when solved separately, but it can become much more difficult to solve both parts together. This is because of the different nature of the two parts. The diffusion equation is a time-dependant partial differential equation, while the reaction term is an ordinary differential equation. Fortunately using operator splitting to solve reaction-diffusion equations gives us the ability to use efficient methods to solve the parts separately, while still giving a good approximation of the solution.

Chai et al. (2013) uses a form of operator splitting called Godunov splitting, or sequential splitting. To explain how this works we simplify the way we write the reaction-diffusion equation. We write

$$\boldsymbol{u}_t = f + g,$$

where f represent the diffusion part of the equation and g represents the reaction part. Normally we would solve the entire equation $f+g$ for the whole interval t_0 to t_1. And then repeat this process for each timestep afterwards. When using operator splitting we instead solve the parts separately. First we solve for f from $\boldsymbol{u}_0$ to a point $\boldsymbol{u}^*$ in the interval of the timestep using the initial value for the equation. Then we solve for g from $\boldsymbol{u}^*$ to $\boldsymbol{u}_1$, but here we use the computed results for f at $\boldsymbol{u}^*$ as the initial value. This will give a pretty good approximation of the solution of the reaction-diffusion equation at $\boldsymbol{u}_1$.

2.4 Implicit Methods

In Chai et al. (2013), the authors chose to use an explicit scheme to solve this problem. This is the most obvious choice as it is very easy to implement and well-suited for solving the problem using parallel computations. As the particular model we are looking at quickly becomes very CPU intensive, we want to take advantage of the ability to use parallel programming. Advantages of using an explicit scheme are that the computational work in each time step is significantly cheaper than a fully implicit scheme, and that they only require nearest-neighbour communication so the parallel scalability is good. A major drawback of using an explicit scheme is that the stability condition puts extremely strong restrictions on Δt.

Advantages of using an implicit scheme are that the stability restriction on Δt disappears. There may still be some restrictions on Δt, but these will be related to accuracy and will be much more lenient than the very

strict condition of the explicit scheme. A disadvantage of using an implicit scheme is that the computation in each time step will be much more costly. This might be alleviated by using an iterative method to solve the implicit scheme instead of using a very computation heavy method like Gaussian elimination. An implicit scheme also requires more communication when it is implemented in parallel.

We want to test whether it is possible to use an implicit scheme and get better computation time than the explicit scheme. An implicit scheme will require more time in each time step and for communication. But hopefully we will be able to show that because we can use a much larger time step and by intelligently programming both the solver and the parallel communication that the implicit method will be more effective than the explicit method currently used to solve this problem.

We are also interested in comparing different types of iterative methods. Some, like the classical iterative methods, are relatively simple methods where each iteration is relatively cheap computation wise, but they usually need a lot of iterations before acceptable convergence has been reached. Another type of iterative method are the more sophisticated methods, like the Conjugate-Gradient method. These methods require a significant amount of computation for each iteration, but convergence is usually relatively fast. Throughout this thesis we will be looking at the performance of classical iterative methods compared to the Conjugate-Gradient method, to see which type of iterative method gives the best results.

Because we use operator splitting we get two separate problems to solve. One partial differential equation, the diffusion equation, and one ordinary differential equation, the reaction equation. The partial differential equation is computationally much heavier to solve than the ordinary differential equation. We will therefore look at ways to efficiently solve the diffusion equation. We will do this by implementing several iterative methods and see if we get an improved performance over the explicit Euler method. The problem we will use for our tests is presented in Chapter 3. At the start we will look at behaviour of the various methods in a serial implementation and for a simpler problem than what they use in Chai et al. (2013). If any of the iterative methods gives promising results in the serial tests we will look at parallel implementation of these methods and see if they still give positive results when parallel communication is taken into account. For both the serial and the parallel implementation we will compare the theoretical and actual results.

2.5 Number of Operations

The computations required for each unknown at each time step is much smaller for the explicit methods than for the implicit methods. Because of this it is obvious that if we use the same time step for the implicit and explicit methods the explicit methods will be much more effective. As such we want to take advantage of the fact that we can use much larger time steps for the implicit schemes as they do not have the same strict stability requirements. We only have to take small enough steps to ensure accuracy. We will now discuss what we can expect for the implicit and explicit schemes if we have the time step Δt for the explicit scheme depend on Δx^2 while the time step for the implicit schemes depends on Δx where we for the three-dimensional space discretization have $\Delta x = \Delta y = \Delta z$. This is why we only need the time step to depend on one of the space discretization parameters. As we are discussing a three dimensional problem we have N^3 unknowns from discretization in space where $N = \frac{1}{\Delta x}$. For the explicit scheme we do in addition get N^2 from the time discretization as $\frac{1}{\Delta t} = \frac{1}{\Delta x^2} = N^2$ while for the implicit scheme we get just N. We also have the computational work that has to be done for each unknown for each time step, which we call C_E for the explicit method and C_I for the implicit method. We now get that the amount of calculation that has to be performed for the explicit scheme is

$$E(\Delta x) = C_E \frac{1}{\Delta x^5} = C_E N^5,$$

while for the implicit scheme it is

$$I(\Delta x) = C_I \frac{1}{\Delta x^4} = C_I N^4.$$

We know that $C_E << C_I$, so for small N we will still have that $E(\Delta x) < I(\Delta x)$, but as N becomes larger the implicit method will eventually overtake the explicit method. While N is small the controlling part of the method is the constants, so as $C_E << C_I$ we get that $E < I$, but as N becomes large the part containing the unknowns will gradually take control and it will become more and more important that the explicit method scales as N^5 while the implicit method just scales as N^4. Eventually N will become so large that $C_I < C_E N$ and therefore $I < E$. When this happens the implicit method will overall be more computationally effective than the explicit method.

Chapter 3

The Diffusion Equation

3.1 Introduction

The diffusion equation is a partial differential equation with first order derivative in time and second order derivatives in space. This equation describes how the concentration of a material changes over time when it is undergoing diffusion. It can generally be written as

$$\frac{\partial \boldsymbol{u}(\boldsymbol{x},t)}{\partial t} = \nabla \cdot (D(\boldsymbol{u},\boldsymbol{x})\nabla \boldsymbol{u}(\boldsymbol{x},t)),$$

where $\boldsymbol{u}(\boldsymbol{x},t)$ is the concentration of the material that is diffusing, in our case calcium, at the time t and the location $\boldsymbol{x}$. $D(\boldsymbol{u},\boldsymbol{x})$ is the diffusion coefficient which varies dependent on the concentration $\boldsymbol{u}$ and the location $\boldsymbol{x}$.

The diffusion equations in the model we discussed in the previous chapter has constant and isotropic diffusion coefficients. Isotropic means that it is equal in all directions. We will therefore focus on this type of diffusion equation. This diffusion equation can be written as

$$\frac{\partial \boldsymbol{u}}{\partial t} = \frac{1}{\beta}\nabla^2 \boldsymbol{u} = \frac{1}{\beta}\Delta \boldsymbol{u},$$

where $\nabla^2 = \Delta$ is the Laplace operator and $\frac{1}{\beta}$ is the diffusion coefficient.

In the three-dimensional case this becomes

$$\boldsymbol{u}_t = \frac{\partial \boldsymbol{u}}{\partial t} = \frac{1}{\beta}\left(\frac{\partial^2 \boldsymbol{u}}{\partial x^2} + \frac{\partial^2 \boldsymbol{u}}{\partial y^2} + \frac{\partial^2 \boldsymbol{u}}{\partial z^2}\right) = \frac{1}{\beta}(\boldsymbol{u}_{xx} + \boldsymbol{u}_{yy} + \boldsymbol{u}_{zz}).$$

3.2 The Test Problem

We will use a diffusion equation with homogeneous Neumann boundary conditions as our test problem. This is the boundary condition used in

the model discussed in Chapter 2. After discretizing this problem on the unit cube using a finite difference scheme in space we will, when using an implicit scheme in time, get a linear system of equations. To solve this equation system, it may be advantageous to use iterative methods instead of a more computationally expensive method like Gaussian elimination. We will get back to this in the following chapter. Such methods may be a good alternative to the computationally attractive, but more unstable Explicit Euler scheme.

The test problem we look at is

$$\begin{aligned} \boldsymbol{u}_t &= \frac{1}{\beta}\nabla^2 \boldsymbol{u} \qquad && t \in (0, T], x, y, z \in \Omega, \\ \frac{\partial \boldsymbol{u}}{\partial n} &= 0 && \text{on } \partial\Omega, \end{aligned}$$

which is a time-dependent, three-dimensional problem in space where $\Omega = [0, 1]^3$ is the unit cube.

3.2.1 The Analytical Solution

We know the general solution to this test problem.

$$\boldsymbol{u} = \sum_{i=0}^{\infty} c_i e^{\frac{-3(\pi i)^2}{\beta}t} \cos(\pi i x) \cos(\pi i y) \cos(\pi i z).$$

It is easy to show that this set of solutions are solutions of our partial differential equation. One simply has to insert it into the equation system and solve.

We see that this gives an infinite set of solutions. To limit this to one specific analytical solution we have make a choice of initial condition, $\boldsymbol{u}_0$. We choose to use

$$\boldsymbol{u}_0 = e^0 \cos(2\pi x) \cos(2\pi y) \cos(2\pi z) = \cos(2\pi x) \cos(2\pi y) \cos(2\pi z).$$

There is now exactly one solution to our test problem. This is

$$\boldsymbol{u} = e^{\frac{-3(2\pi)^2}{\beta}t} \cos(2\pi x) \cos(2\pi y) \cos(2\pi z).$$

3.2.2 Discretization in Space

The purpose of this test problem is not simply to solve it. What we are after is testing the various methods described in Chapter 4. We can do this by using these methods when solving the test problem numerically and then as we know the exact solution we can see how the error behaves when comparing this exact solution to the numerical solutions we obtain from the

various methods.

To solve this problem numerically we first need to discretize the problem in space. The easiest way to do this is by using a second order finite difference scheme.

$$\begin{aligned}
\boldsymbol{u}_t &= \frac{1}{\beta}(\boldsymbol{u}_{xx} + \boldsymbol{u}_{yy} + \boldsymbol{u}_{zz}) \\
\boldsymbol{u}_t &= \frac{1}{\beta}\left(\frac{\boldsymbol{u}_{i+1,j,k} - 2\boldsymbol{u}_{i,j,k} + \boldsymbol{u}_{i-1,j,k}}{h^2} + \frac{\boldsymbol{u}_{i,j+1,k} - 2\boldsymbol{u}_{i,j,k} + \boldsymbol{u}_{i,j-1,k}}{h^2}\right. \\
&\quad \left. + \frac{\boldsymbol{u}_{i,j,k+1} - 2\boldsymbol{u}_{i,j,k} + \boldsymbol{u}_{i,j,k-1}}{h^2}\right) \\
\boldsymbol{u}_t &= \frac{1}{\beta h^2}\left(\boldsymbol{u}_{i+1,j,k} + \boldsymbol{u}_{i-1,j,k} + \boldsymbol{u}_{i,j+1,k} + \boldsymbol{u}_{i,j-1,k} + \boldsymbol{u}_{i,j,k+1} + \boldsymbol{u}_{i,j,k-1} - 6\boldsymbol{u}_{i,j,k}\right).
\end{aligned}$$

It is generally useful to look at these kinds of problems in matrix form. We can obviously write this set of equations as

$$\boldsymbol{u}_t = \frac{1}{\beta h^2}\boldsymbol{A}\boldsymbol{u},$$

where $\boldsymbol{A}$ is the matrix for the discretization in space. This matrix will have -6 on the diagonal. A band of ones on each side of the diagonal, which are the points on each side in the x-direction. There will be another band of ones N points out from the diagonal. These ones are the points in the y-direction and N is the number of nodes used to discretize the problem in x-direction. There will be one last band of ones that is N^2 points out from the diagonal. This band is the discretization in the z-direction. This is assuming we use the same discretization in all directions in space. This will result in a problem with a matrix $\boldsymbol{A}$ which is $N^3 \times N^3$ large, where N is the number of nodes in each direction. This will give us a system with N^3 unknowns.

While considering theoretical problems or doing calculations we do not need to give this matrix much thought. We know what it looks like and that is enough. However when we write computer programs to help us solve the numerical problems it is useful to consider how to generate this matrix as well as what the final result is. There are of course obvious choices like using a double `for`-loop to loop through each element or the slightly more effective way of using a single `for`-loop and assigning all elements in each row as we know where the non-zero elements are located based on the discretization used. This will generally be the best way of implementing the matrix when using a programming language which is not optimized for matrix and vector operations, for instance C or C++. However when we are using a programming language like MATLAB which has support for vectorization, it is better to take advantage of this optimization. Then one way of

generating the matrix is by using the Kronecker product. This is a tensor product for matrices which result in a block matrix defined by

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{bmatrix} a_{11}\boldsymbol{B} & \cdots & a_{1n}\boldsymbol{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\boldsymbol{B} & \cdots & a_{mn}\boldsymbol{B} \end{bmatrix}$$

where $\boldsymbol{A}$ is an $m \times n$ matrix and $\boldsymbol{B}$ is a $p \times q$ matrix resulting in the Kronecker product $\boldsymbol{A} \otimes \boldsymbol{B}$ producing a $mp \times nq$ block matrix. More information on this product can be found in Chapter 3 in Lyche (2013). This product is relevant for us because we can use it to generate our matrix. The way this can be done is by first defining the much smaller matrices for the discretization in each direction. These matrices will be identical and will be $N \times N$ large with -2 on the diagonal and 1 on each side next to diagonal, we call this matrix $\boldsymbol{T}$. By using Kronecker products with the identity matrix, $\boldsymbol{I}$, we can generate the larger matrix that contains the discretization in all three directions. The sum of the Kronecker products

$$\boldsymbol{A} = \boldsymbol{I} \otimes (\boldsymbol{I} \otimes \boldsymbol{T}) + \boldsymbol{I} \otimes (\boldsymbol{T} \otimes \boldsymbol{I}) + (\boldsymbol{T} \otimes \boldsymbol{I}) \otimes \boldsymbol{I},$$

results in the matrix $\boldsymbol{A}$ we want with -6 on the diagonal and three correctly placed bands of ones.

The truncation error for this discretization scheme in space is

$$\begin{aligned} R_{i,j,k} = &\frac{1}{12}\boldsymbol{u}_{e,xxxx}(x_i, y_j, z_k)\Delta x^2 + \mathcal{O}(\Delta x^4) + \frac{1}{12}\boldsymbol{u}_{e,yyyy}(x_i, y_j, z_k)\Delta y^2 + \mathcal{O}(\Delta y^4) \\ &+ \frac{1}{12}\boldsymbol{u}_{e,zzzz}(x_i, y_j, z_k)\Delta z^2 + \mathcal{O}(\Delta z^4). \end{aligned}$$

When using the same discretzation in each direction we get $\Delta x = \Delta y = \Delta z = h$ which gives us

$$R_{i,j,k} = \frac{h^2}{12}\left(\boldsymbol{u}_{e,xxxx}(x_i, y_j, z_k) + \boldsymbol{u}_{e,yyyy}(x_i, y_j, z_k) + \boldsymbol{u}_{e,zzzz}(x_i, y_j, z_k)\right) + \mathcal{O}(h^4).$$

Langtangen (2014) gives more information on how this truncation error is derived and what the usefulness of it is. This error is useful to us because it shows that this is a second order scheme and as such we can expect second order convergence in space. This is of course a truth which needs some modification, because the convergence of this scheme is also dependent on the order of the discretization in time. So if this scheme is paired with a first order scheme in time, like those in Section 3.2.3, the convergence will be first order and not second order even though the space discretization is second order. This is because the part of the discretization with the lowest order always is the governing part when it comes to rate of convergence.

3.2.3 Discretization in Time

While the choice of discretization scheme in space is rather obvious, as we need a second order scheme to discretize the second order derivatives, the choice of discretization in time offer several options. Here we will look at using the classical schemes Backward Euler and Crank-Nicolson. We will also mention Forward Euler as this scheme is interesting for comparison purposes.

Forward Euler

The discretization scheme for Forward Euler is:

$$\begin{aligned}\frac{\boldsymbol{u}^{n+1}-\boldsymbol{u}^n}{\Delta t} &= \frac{1}{\beta h^2}\boldsymbol{A}\boldsymbol{u}^n \\ \boldsymbol{u}^{n+1} &= \boldsymbol{u}^n + \frac{\Delta t}{\beta h^2}\boldsymbol{A}\boldsymbol{u}^n \\ \boldsymbol{u}^{n+1} &= (\boldsymbol{I} + \frac{\Delta t}{\beta h^2}\boldsymbol{A})\boldsymbol{u}^n.\end{aligned}$$

We see that this scheme does not yield a set of linear equations. It is an explicit scheme and as such it can be solved directly. It is therefore a useless choice when we want to test our iterative methods from Chapter 4. It can however be very useful to compare the results from this method with the results from implicit schemes solved by our iterative methods to see how the iterative methods holds up.

The advantage of the forward Euler scheme is that it can be solved directly. The disadvantage is that it has very strict stability conditions. An analysis of the stability criteria of the Forward Euler scheme can be found in Tveito and Winther (2009). This book uses a simple one-dimensional example, but it is easy to extend the analysis to multiple dimensions.

This scheme is a first order scheme with the truncation error R^n

$$R^n = \frac{1}{2}\boldsymbol{u}_{e,tt}(t_n)\Delta t + \mathcal{O}(\Delta t^2).$$

This error only includes the error for the time discretization. The truncation error when we include the second order space discretization is

$$\begin{aligned}R^n_{i,j,k} = &\frac{1}{2}\boldsymbol{u}_{e,tt}(x_i,y_j,z_k,t_n)\Delta t - \frac{1}{12\beta}\boldsymbol{u}_{e,xxxx}(x_i,y_j,z_k,t_n)\Delta x^2 \\ &- \frac{1}{12\beta}\boldsymbol{u}_{e,yyyy}(x_i,y_j,z_k,t_n)\Delta y^2 - \frac{1}{12\beta}\boldsymbol{u}_{e,zzzz}(x_i,y_j,z_k,t_n)\Delta z^2 \\ &+ \mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta x^4) + \mathcal{O}(\Delta y^4) + \mathcal{O}(\Delta z^4).\end{aligned}$$

When we use the same number of nodes in each direction we get $\Delta x = \Delta y = \Delta z = h$ which gives us

$$\begin{aligned} R^n_{i,j,k} = &\frac{1}{2}\boldsymbol{u}_{e,tt}(x_i, y_j, z_k, t_n)\Delta t - \frac{h^2}{12\beta}(\boldsymbol{u}_{e,xxxx}(x_i, y_j, z_k, t_n) \\ &+ \boldsymbol{u}_{e,yyyy}(x_i, y_j, z_k, t_n) + \boldsymbol{u}_{e,zzzz}(x_i, y_j, z_k, t_n)) \\ &+ \mathcal{O}(\Delta t^2) + \mathcal{O}(h^4). \end{aligned}$$

We see that this truncation error is simply adding the truncation error from the space discretization to the error for the time discretization scheme. This is still a first order scheme. We see this because the first term in the truncation error contains Δt. This term will govern the error because it will become smaller at a much slower rate than the other terms.

Backward Euler

The discretization scheme for Backward Euler is:

$$\begin{aligned} \frac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^n}{\Delta t} &= \frac{1}{\beta h^2}\boldsymbol{A}\boldsymbol{u}^{n+1} \\ \boldsymbol{u}^{n+1} - \frac{\Delta t}{\beta h^2}\boldsymbol{A}\boldsymbol{u}^{n+1} &= \boldsymbol{u}^n \\ (\boldsymbol{I} - \frac{\Delta t}{\beta h^2}\boldsymbol{A})\boldsymbol{u}^{n+1} &= \boldsymbol{u}^n. \end{aligned}$$

This scheme is implicit so it results in a system of linear equations that we need to solve. This is perfect for our purposes as we can use this method to solve our test problem by using the iterative methods we want to test. Another advantage of using the Backward Euler scheme for discretizing in time is that it is unconditionally stable. This can be shown by using Von Neumann analysis. A one-dimensional example of this can be found in Tveito and Winther (2009). The example in this book is simpler than our test problem, but the analysis can easily be extended to more difficult problems.

The truncation error, R^n, for the time discretization scheme for Backward Euler scheme is

$$R^n = -\frac{1}{2}\boldsymbol{u}_{e,tt}(t_n)\Delta t + \mathcal{O}(\Delta t^2),$$

while if we include the truncation error for the space discretization as well we get

$$\begin{aligned} R^n_{i,j,k} = &-\frac{1}{2}\boldsymbol{u}_{e,tt}(x_i, y_j, z_k, t_n)\Delta t - \frac{1}{12\beta}\boldsymbol{u}_{e,xxxx}(x_i, y_j, z_k, t_n)\Delta x^2 \\ &- \frac{1}{12\beta}\boldsymbol{u}_{e,yyyy}(x_i, y_j, z_k, t_n)\Delta y^2 - \frac{1}{12\beta}\boldsymbol{u}_{e,zzzz}(x_i, y_j, z_k, t_n)\Delta z^2 \\ &+ \mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta x^4) + \mathcal{O}(\Delta y^4) + \mathcal{O}(\Delta z^4). \end{aligned}$$

When using the same discretzation in each spatial direction we get $\Delta x = \Delta y = \Delta z = h$ which gives us

$$\begin{aligned} R^n_{i,j,k} = &-\frac{1}{2}\boldsymbol{u}_{e,tt}(x_i, y_j, z_k, t_n)\Delta t - \frac{h^2}{12\beta}(\boldsymbol{u}_{e,xxxx}(x_i, y_j, z_k, t_n)\Delta x^2 \\ &+ \boldsymbol{u}_{e,yyyy}(x_i, y_j, z_k, t_n) + \boldsymbol{u}_{e,zzzz}(x_i, y_j, z_k, t_n)) \\ &+ \mathcal{O}(\Delta t^2) + \mathcal{O}(h^4). \end{aligned}$$

We see from this truncation error that the Backward Euler scheme also is a first order scheme, the same as Forward Euler. This is of course expected as the only difference between them is that in Forward Euler the right hand side of the equation is evaluated in the previous time step, while for Backward Euler it is evaluated for the current time step.

Crank-Nicolson

This is a slightly more advanced scheme than the two others we have described. The Crank-Nicolson scheme is defined by

$$\begin{aligned} \frac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^n}{\Delta t} &= \frac{1}{2}\left(\frac{1}{\beta h^2}\boldsymbol{A}\boldsymbol{u}^{n+1} + \frac{1}{\beta h^2}\boldsymbol{A}\boldsymbol{u}^n\right) \\ \boldsymbol{u}^{n+1} - \boldsymbol{u}^n &= \frac{\Delta t}{2\beta h^2}\boldsymbol{A}\boldsymbol{u}^{n+1} + \frac{\Delta t}{2\beta h^2}\boldsymbol{A}\boldsymbol{u}^n \\ \boldsymbol{u}^{n+1} - \frac{\Delta t}{2\beta h^2}\boldsymbol{A}\boldsymbol{u}^{n+1} &= \boldsymbol{u}^n + \frac{\Delta t}{2\beta h^2}\boldsymbol{A}\boldsymbol{u}^n \\ (\boldsymbol{I} - \frac{\Delta t}{2\beta h^2}\boldsymbol{A})\boldsymbol{u}^{n+1} &= (\boldsymbol{I} + \frac{\Delta t}{2\beta h^2}\boldsymbol{A})\boldsymbol{u}^n. \end{aligned}$$

We see that this scheme also is an implicit scheme so we have to solve a system of equations again.

The truncation error when only looking at the time discretization part of scheme is

$$R^{n+\frac{1}{2}} = \left(\frac{1}{24}\boldsymbol{u}_{e,ttt}\left(t_{n+\frac{1}{2}}\right) + \frac{1}{8}\boldsymbol{u}_{e,tt}(t_n)\right)\Delta t^2 + \mathcal{O}(\Delta t^4).$$

While the truncation error when we include the space discretization as well is

$$\begin{aligned} R^{n+\frac{1}{2}}_{i,j,k} = &\frac{1}{24}\boldsymbol{u}_{e,ttt}(x_i, y_j, z_k, t_n)\Delta t^2 \\ &+ \frac{1}{8}\boldsymbol{u}_{e,ttxx}(x_i, y_j, z_k, t_n)\Delta t^2 + \frac{1}{12}\boldsymbol{u}_{e,xxxx}(x_i, y_j, z_k, t_n)\Delta x^2 \\ &+ \frac{1}{8}\boldsymbol{u}_{e,ttyy}(x_i, y_j, z_k, t_n)\Delta t^2 + \frac{1}{12}\boldsymbol{u}_{e,yyyy}(x_i, y_j, z_k, t_n)\Delta y^2 \\ &+ \frac{1}{8}\boldsymbol{u}_{e,ttzz}(x_i, y_j, z_k, t_n)\Delta t^2 + \frac{1}{12}\boldsymbol{u}_{e,zzzz}(x_i, y_j, z_k, t_n)\Delta z^2 \\ &+ \mathcal{O}\left(\Delta t^4\right) + \mathcal{O}\left(\Delta x^4\right) + \mathcal{O}\left(\Delta y^4\right) + \mathcal{O}\left(\Delta z^4\right) \\ &+ \mathcal{O}\left(\Delta t^2\Delta x^2\right) + \mathcal{O}\left(\Delta t^2\Delta y^2\right) + \mathcal{O}\left(\Delta t^2\Delta z^2\right). \end{aligned}$$

When we have the same number of nodes in each direction we get $\Delta x = \Delta y = \Delta z = h$ which gives us

$$\begin{aligned} R_{i,j,k}^{n+\frac{1}{2}} &= \frac{1}{24}\boldsymbol{u}_{e,ttt}(x_i, y_j, z_k, t_n)\Delta t^2 \\ &+ \frac{\Delta t^2}{8}\left(\boldsymbol{u}_{e,ttxx}(x_i, y_j, z_k, t_n) + \boldsymbol{u}_{e,ttyy}(x_i, y_j, z_k, t_n) + \boldsymbol{u}_{e,ttzz}(x_i, y_j, z_k, t_n)\right) \\ &+ \frac{h^2}{12}\left(\boldsymbol{u}_{e,xxxx}(x_i, y_j, z_k, t_n) + \boldsymbol{u}_{e,yyyy}(x_i, y_j, z_k, t_n) + \boldsymbol{u}_{e,zzzz}(x_i, y_j, z_k, t_n)\right) \\ &+ \mathcal{O}\left(\Delta t^4\right) + \mathcal{O}\left(h^4\right) + \mathcal{O}\left(\Delta t^2 h^2\right). \end{aligned}$$

As we can see from this, the Crank-Nicolson scheme is a second order scheme. This means that this scheme will converge at a rate of Δt^2 instead of Δt as the case is for the other schemes. Therefore this scheme is more accurate and gives faster convergence towards the exact solution than the first order schemes.

The disadvantage with Crank-Nicolson compared to Backward Euler is that it is prone to oscillations. As such there is a stability condition for this scheme, such as for Forward Euler. Because of this it might give better results using the unconditionally stable Backward Euler scheme instead of the on paper more accurate Crank-Nicolson scheme. This will of course depend on how strict the stability condition for this scheme needs to be. This stability condition can be as strict as the stability condition for Forward Euler or much more relaxed. The method can even be unconditionally stable. This depends on the problem we want to solve. To find the stability condition for a specific problem one can, as for Forward Euler, use Von Neumann stability analysis

Chapter 4

Classical Iterative Methods

4.1 Introduction

The mathematical theory and methods described in this chapter is mostly based on Lyche (2013), Saad (2003) and Mardal and Logg (2013).

In this chapter we will look at methods for solving a general set of linear equations. This theory can be applied to solving partial differential equations, as discussed in Chapter 3. Therefore it is useful to have efficient ways to solve linear systems of equations when solving partial differential equations with implicit methods. To solve

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b},$$

where $\boldsymbol{x}$ is very large, is useful for solving partial differential equations. We can solve this by using direct methods like Gaussian elimination or Cholesky factorization. As long as there are no rounding errors these methods will find the exact solution in a finite number of operations. These methods require a lot of operations and they always have to do the complete calculation regardless of how the matrix looks. This is a disadvantage. Because of this, these methods usually use much more time than iterative methods. The iterative methods usually use a much smaller number of operations and require much less storage. Hence iterative methods are very good for solving large sparse systems. The denser the matrix the closer the solving time and storage required gets to what the direct methods require.

In iterative methods we approximate the solution $\boldsymbol{x}$ by $\boldsymbol{x}_k$ and iterate so each successive iteration $\boldsymbol{x}^{(k)}$ gets closer to the exact solution. We start by approximating $\boldsymbol{x}$ by $\boldsymbol{x}^{(0)}$. We then iterate so we compute a sequence of approximated solutions $\{\boldsymbol{x}^{(k)}\}$ such that $\boldsymbol{x}^{(k)} \to \boldsymbol{x}$. How the iterations are carried out depends on which iterative method we use. This chapter will describe some iterative methods. We will introduce the fixed point form of $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. This is needed for explaining the component form of some of the methods.

As the iterative methods use iterations to get an increasingly better approximation of the exact solution, we need to know that they will actually converge. The arguments for ensuring convergence for the Jacobi and Gauss-Seidel method are very similar. As such we present the conditions for convergence in the subsections for each method, but the argument for why these are the necessary conditions are presented in a separate section later in this chapter. The arguments for convergence of Richardson's method are somewhat different so we will present this in a separate section.

4.2 Fixed Point Form

All we do to obtain this form is to look at the i-th equation and solve for $\boldsymbol{x}_i = \boldsymbol{x}(i)$:

$$\sum_{j=1}^{n} a_{ij}\boldsymbol{x}_j = \boldsymbol{b}_i$$

We isolate the i-th component

$$a_{ii}\boldsymbol{x}_i + \sum_{j=1}^{i-1} a_{ij}\boldsymbol{x}_j + \sum_{j=i+1}^{n} a_{ij}\boldsymbol{x}_j = b_i$$

and solve for $\boldsymbol{x}_i$

$$\boldsymbol{x}_i = \frac{1}{a_{ii}}\left(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}\boldsymbol{x}_j\right) \quad \text{for} \quad i = 1, 2, \ldots, n.$$

The formula shows that any method that require this form to get the iterative method comes with the requirement that all the diagonal elements are non-zero.

4.3 Richardson Iteration

This method is referred to either as Richardson's method or as Richardson iteration. It is the simplest of the classical iterative methods. Here we make use of the residual vector

$$\boldsymbol{r}^{(k)} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)}.$$

As this residual vector represents the error for our current approximation of $\boldsymbol{x}$ we can use this to construct the next approximation. We do this by simply adding or subtracting this residual from our current approximation. Lewis Richardson looked at the iteration where one simply adds the residual so originally this method was

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + (\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)}).$$

Later a parameter has been introduced to ensure that the method converges, or in cases where we are guaranteed convergence, it can be used to speed up convergence. This gives us the method

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)}).$$

This is the matrix form of Richardson's method. One can also look at the case where we subtract the residual instead. Using this method simply requires a different choice of α, but is otherwise identical to the version where we add the residual.

The equivalent component form is

$$\boldsymbol{x}_i^{(k+1)} = \boldsymbol{x}_i^{(k)} + \alpha \boldsymbol{r}_i^{(k)}, \quad \boldsymbol{r}_i^{(k)} = b_i - \sum_{j=1}^{n} a_{ij}\boldsymbol{x}_j^{(k)}, \quad \text{for} \quad i = 1, 2, \ldots, n.$$

There are no fundamental differences between the component and matrix form of this method. These forms are simply two ways to express the same method.

4.3.1 Convergence

To find the properties needed to guarantee convergence, we look at the error for the Richardson iteration.

The iteration error can generally be expressed as

$$\boldsymbol{e}^{(k)} = \boldsymbol{x}^{(k)} - \boldsymbol{x}.$$

Inserting this into the Richarson iteration and subtracting $\boldsymbol{x}$ from both sides gives us

$$\boldsymbol{e}^{(k+1)} = \boldsymbol{e}^{(k)} + \alpha \boldsymbol{A}\boldsymbol{e}^{(k)}.$$

Using a norm to quantify the error we get

$$\begin{aligned} ||\boldsymbol{e}^{(k+1)}|| &= ||\boldsymbol{e}^{(k)} + \alpha \boldsymbol{A}\boldsymbol{e}^{(k)}|| \\ &\leq ||\boldsymbol{I} + \alpha \boldsymbol{A}||||\boldsymbol{e}^{(k)}||. \end{aligned}$$

We see from this that if $||\boldsymbol{I} + \alpha \boldsymbol{A}|| < 1$ then the iteration will converge. The parameter α can be chosen so this convergence condition can be fulfilled.

4.4 Jacobi's method

This method uses the fixed point form we discussed in Section 4.2 to solve for the next step $\boldsymbol{x}^{(k+1)}$ instead of adding the residual, as was done in Richardson' method. The component form of this method is

$$\boldsymbol{x}_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} \boldsymbol{x}_j^{(k)} \right) \quad \text{for} \quad i = 1, 2, \ldots, n.$$

To obtain the matrix form for this method we decompose $\boldsymbol{A}$ into a matrix $\boldsymbol{D}$ with its diagonal components and a matrix $\boldsymbol{R}$ containing the remainder of the matrix. We now see that if we just assemble all the component forms we will get these two matrices and this gives us the matrix form

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{R}\boldsymbol{x}^{(k)}),$$

which is a more compact form for describing the method.

4.4.1 Convergence

We are guaranteed that the Jacobi method converges if the spectral radius of the iteration matrix, $\boldsymbol{D}^{-1}\boldsymbol{R}$ is less than 1:

$$\rho(\boldsymbol{D}^{-1}\boldsymbol{R}) < 1.$$

This is the standard convergence condition for any iterative method. This will be discussed further in Section 4.6. The method will also converge if the matrix $\boldsymbol{A}$ is strictly or irreducibly diagonally dominant. A matrix is irreducibly diagonally dominant if it is irreducible, meaning that it is not similar to a block upper triangular matrix, and it is weakly diagonally dominant with at least one row that is strictly diagonally dominant. It is possible for the Jacobi method to converge even if this is not fulfilled, but we can not guarantee convergence in these cases.

4.4.2 Weighted Jacobi's method

There is a version of Jacobi's method called the Weighted Jacobi method which introduces a weight parameter ω.

$$\boldsymbol{x}^{(k+1)} = \omega \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{R}\boldsymbol{x}^{(k)}) + (1 - \omega)\boldsymbol{x}^{(k)}.$$

As explained in Section 13.2.2 in Saad (2003), this method does not work to accelerate Jacobi's method. Its function is primarily as a smoother. As this version does not accelerate Jacobi's iteration we will get the fastest results when using $\omega = 1$. However when we are using it as a smoother the optimal value of ω is $\omega = \frac{2}{3}$. The reason for this is discussed in Saad (2003).

We include this method as it will be interesting to test whether our test problem benefits from using a smoother.

The Weighted Jacobi method has the same convergence criteria as the regular Jacobi's method.

4.5 Gauss-Seidel's method

This method is a refinement of Jacobi's method where we exploit the fact that some of the values of $\boldsymbol{x}^{(k+1)}$ have already been calculated. For the component form of this method we know that when we want to calculate $\boldsymbol{x}_i^{(k+1)}$ the first $i-1$ values of $\boldsymbol{x}^{(k+1)}$ have already been calculated so we can use this in calculating the i-th value to get a more exact approximation of $\boldsymbol{x}$ faster than the Jacobi method gives us.

Using what we have now described, the component form of Gauss-Seidel's method becomes

$$\boldsymbol{x}_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} \boldsymbol{x}_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} \boldsymbol{x}_j^{(k)} \right) \quad \text{for} \quad i = 1, 2, \ldots, n.$$

To get the matrix form for this method we once again simply assemble the separate component forms for each $\boldsymbol{x}_i^{(k+1)}$ to get the matrices. We only use the values of $\boldsymbol{x}^{(k)}$ where a corresponding value of $\boldsymbol{x}^{(k+1)}$ has not yet been generated. We see that by decomposing $\boldsymbol{A}$ into a matrix $\boldsymbol{D}$ containing the diagonal elements, a matrix $\boldsymbol{L}$ containing the lower triangular elements and a matrix $\boldsymbol{U}$ containing the upper triangular elements of $\boldsymbol{A}$, that the system

$$(\boldsymbol{D} + \boldsymbol{L})\boldsymbol{x}^{(k+1)} = \boldsymbol{b} - \boldsymbol{U}\boldsymbol{x}^{(k)},$$

corresponds to the component form of the Gauss-Seidel method. To get $\boldsymbol{x}^{(k+1)}$ alone on the left hand side we simply multiply both sides with the inverse of $(\boldsymbol{D} + \boldsymbol{L})$. This gives us the matrix form

$$\boldsymbol{x}^{(k+1)} = (\boldsymbol{D} + \boldsymbol{L})^{-1}(\boldsymbol{b} - \boldsymbol{U}\boldsymbol{x}^{(k)}).$$

4.5.1 Convergence

Whether the Gauss-Seidel method converges or not depends on the properties of the matrix $\boldsymbol{A}$. If either of the following are true for $\boldsymbol{A}$ then Gauss-Seidel converges.

- $\boldsymbol{A}$ is symmetric positive definite.
- $\boldsymbol{A}$ is strictly or irreducibly diagonally dominant.

If either of these conditions are fulfilled we are guaranteed that this method converges. It is of course possible that the method converges for some problems that do not have either of these properties, but if these are not fulfilled we cannot guarantee convergence.

4.6 Convergence of Jacobi's and Gauss-Seidel's method

As we see from Section 4.4, Jacobi's method can be written as

$$\boldsymbol{x}^{(k+1)} = -\boldsymbol{D}^{-1}\boldsymbol{R}\boldsymbol{x}^{(k)} + \boldsymbol{D}^{-1}\boldsymbol{b},$$

and from Section 4.5 we see that Gauss-Seidel's method can be written as

$$\boldsymbol{x}^{(k+1)} = -(\boldsymbol{D}+\boldsymbol{L})^{-1}\boldsymbol{U}\boldsymbol{x}^{(k)} + (\boldsymbol{D}+\boldsymbol{L})^{-1}\boldsymbol{b}.$$

This can generally be written as

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{G}\boldsymbol{x}^{(k)} + \boldsymbol{c},$$

where for Jacobi we have

$$\boldsymbol{G} = -\boldsymbol{D}^{-1}\boldsymbol{R} \quad \text{and} \quad \boldsymbol{c} = \boldsymbol{D}^{-1}\boldsymbol{b},$$

and for Gauss-Seidel we have

$$\boldsymbol{G} = -(\boldsymbol{D}+\boldsymbol{L})^{-1}\boldsymbol{U} \quad \text{and} \quad \boldsymbol{c} = (\boldsymbol{D}+\boldsymbol{L})^{-1}\boldsymbol{b}.$$

4.6.1 Condition for Convergence for the Iteration Matrix

We will now derive the condition for convergence on this general expression. We do this by subtracting the exact solution $\boldsymbol{x} = \boldsymbol{G}\boldsymbol{x} + \boldsymbol{c}$ from the iterative solution $\boldsymbol{x}^{(k+1)} = \boldsymbol{G}\boldsymbol{x}^{(k)} + \boldsymbol{c}$. We get

$$\begin{aligned}
\boldsymbol{x}^{(k+1)} - \boldsymbol{x} &= \boldsymbol{G}\boldsymbol{x}^{(k)} + \boldsymbol{c} - (\boldsymbol{G}\boldsymbol{x} + \boldsymbol{c}) \\
\boldsymbol{x}^{(k+1)} - \boldsymbol{x} &= \boldsymbol{G}\boldsymbol{x}^{(k)} - \boldsymbol{G}\boldsymbol{x} + \boldsymbol{c} - \boldsymbol{c} \\
\boldsymbol{x}^{(k+1)} - \boldsymbol{x} &= \boldsymbol{G}(\boldsymbol{x}^{(k)} - \boldsymbol{x}).
\end{aligned}$$

We now use induction to show the next part.
We start by looking at the expression for $k = 0$ and $k = 1$

$$
\begin{aligned}
&k = 0.\\
&\quad \boldsymbol{x}^{(1)} - \boldsymbol{x} = \boldsymbol{G}(\boldsymbol{x}^{(0)} - \boldsymbol{x})\\
&\qquad \boldsymbol{x}^{(1)} = \boldsymbol{G}(\boldsymbol{x}^{(0)} - \boldsymbol{x}) + \boldsymbol{x}.
\end{aligned}
$$

$$
\begin{aligned}
&k = 1.\\
&\quad \boldsymbol{x}^{(2)} - \boldsymbol{x} = \boldsymbol{G}(\boldsymbol{x}^{(1)} - \boldsymbol{x})\\
&\quad \boldsymbol{x}^{(2)} - \boldsymbol{x} = \boldsymbol{G}((\boldsymbol{G}(\boldsymbol{x}^{(0)} - \boldsymbol{x}) + \boldsymbol{x}) - \boldsymbol{x})\\
&\quad \boldsymbol{x}^{(2)} - \boldsymbol{x} = \boldsymbol{G}(\boldsymbol{G}(\boldsymbol{x}^{(0)} - \boldsymbol{x})\\
&\quad \boldsymbol{x}^{(2)} - \boldsymbol{x} = \boldsymbol{G}^2(\boldsymbol{x}^{(0)} - \boldsymbol{x}).
\end{aligned}
$$

We now assume that the expression

$$\boldsymbol{x}^{(k-1)} - \boldsymbol{x} = \boldsymbol{G}^{k-1}(\boldsymbol{x}^{(0)} - \boldsymbol{x}),$$

holds so we just need to show that it also holds for k

$$
\begin{aligned}
\boldsymbol{x}^{(k)} - \boldsymbol{x} &= \boldsymbol{G}(\boldsymbol{x}^{(k-1)} - \boldsymbol{x})\\
\boldsymbol{x}^{(k)} - \boldsymbol{x} &= \boldsymbol{G}(\boldsymbol{G}^{k-1}(\boldsymbol{x}^{(0)} - \boldsymbol{x}) + \boldsymbol{x} - \boldsymbol{x})\\
\boldsymbol{x}^{(k)} - \boldsymbol{x} &= \boldsymbol{G}^{k}(\boldsymbol{x}^{(0)} - \boldsymbol{x}).
\end{aligned}
$$

From this we clearly see that $\boldsymbol{x}^{(k)} - \boldsymbol{x} \to 0$ if $\boldsymbol{G}^k \to 0$. As we want to show that $\boldsymbol{x}^{(k+1)} = \boldsymbol{G}\boldsymbol{x}^{(k)} + \boldsymbol{c}$ converges if and only if $\lim_{k\to\infty} \boldsymbol{G}^k = 0$ we also have to show the converse, meaning that we have to show that if $\boldsymbol{x}^{(k)} - \boldsymbol{x}$ converges to zero then $\boldsymbol{G}^k \to 0$ as $k \to \infty$. This can be shown by choosing $\boldsymbol{x}^{(0)} - \boldsymbol{x} = \boldsymbol{e}_j$ where $\boldsymbol{e}_j$ is the j-th unit vector for $j = 1, \ldots, n$. If we do this we get

$$
\begin{aligned}
\boldsymbol{x}^{(k)} - \boldsymbol{x} &= \boldsymbol{G}^x(\boldsymbol{x}^{(0)} - \boldsymbol{x})\\
\boldsymbol{x}^{(k)} - \boldsymbol{x} &= \boldsymbol{G}^x\boldsymbol{e}_j\\
\boldsymbol{x}^{(k)} - \boldsymbol{x} &= \boldsymbol{G}^k_j.
\end{aligned}
$$

As we now have that the left hand side of this equation converges to zero we must also have that the jth column of $\boldsymbol{G}^k$ goes to zero. As this equation must hold for all $j = 1, \ldots, n$ we get that each column of $\boldsymbol{G}^k$ approaches 0 as $k \to \infty$ because $\boldsymbol{x}^{(k)} - \boldsymbol{x}$ is converging.

From this we see that the iterative method $\boldsymbol{x}^{(k+1)} = \boldsymbol{G}\boldsymbol{x}^{(k)} + \boldsymbol{c}$ converges if and only if $\lim_{k\to\infty} \boldsymbol{G}^k = 0$ so this is a necessary and sufficient condition for convergence. We see that this condition can be applied to both Jacobi and Gauss-Seidel. We just need to use the appropriate iteration matrix for $\boldsymbol{G}$.

Using this result we can derive another sufficient condition for convergence by using a consistent matrix norm. We get

$$\begin{aligned} ||\boldsymbol{x}^{(k)} - \boldsymbol{x}|| &= ||\boldsymbol{G}^k(\boldsymbol{x}^{(0)} - \boldsymbol{x})|| \\ &\leq ||\boldsymbol{G}^k||||\boldsymbol{x}^{(0)} - \boldsymbol{x}|| \\ &\leq ||\boldsymbol{G}||^k||\boldsymbol{x}^{(0)} - \boldsymbol{x}||. \end{aligned}$$

This does obviously converge towards zero if $||\boldsymbol{G}|| < 1$ so as long as this is true we are guaranteed convergence. This argument and the condition for convergence derived from it strongly resembles the arguments used to guarantee convergence for Richardson's method.

While the condition $\lim_{k\to\infty} \boldsymbol{G}^k = 0$ is all we need to guarantee convergence it is a condition that can be difficult to use. Fortunately there is an equivalent condition that we can use instead. It is possible to show that for any $\boldsymbol{G} \in \mathbb{C}^{n\times n}$ we have

$$\lim_{k\to\infty} \boldsymbol{G}^k = 0 \Leftrightarrow \rho(\boldsymbol{G}) < 1,$$

where $\rho(\boldsymbol{G})$ is the spectral radius of $\boldsymbol{G}$, defined by $\rho(\boldsymbol{G}) = \max_{\mu\in\sigma(\boldsymbol{G})} |\mu|$. This proof follows Section 8.4.1 in Lyche (2013) and we will now take some time to present it here. To show that this is true we need to show that $\rho(\boldsymbol{G}) < 1$ is both a necessary and a sufficient condition for $\lim_{k\to\infty} \boldsymbol{G}^k = 0$.

It is easy to show that it is a necessary condition. This follows directly from the properties of eigenpairs. If we have an eigenpair, $(\mu, \boldsymbol{w})$, of $\boldsymbol{G}$ with the properties $|\mu| \geq 1$ and $||\boldsymbol{w}||_2 = 1$, then from $\boldsymbol{G}^k\boldsymbol{w} = \mu^k\boldsymbol{w}$ we get

$$||\boldsymbol{G}^k||_2 \geq ||\boldsymbol{G}^k\boldsymbol{w}||_2 = ||\mu^k\boldsymbol{w}||_2 = |\mu|^k,$$

and from this it obviously follows that $\boldsymbol{G}^k$ does not go to zero. So for $\lim_{k\to\infty} \boldsymbol{G}^k = 0$ it is necessary to have $\rho(\boldsymbol{G}) < 1$.

Now we just need to show that this is a sufficient condition as well as a necessary one. This is more difficult to show and we need to do this in stages.

We start by showing that $\rho(\boldsymbol{G}) \leq ||\boldsymbol{G}||$. We use that $(\mu, \boldsymbol{w})$ is an eigenpair of $\boldsymbol{G}$ and we define $\boldsymbol{W} = [\boldsymbol{w}, \ldots, \boldsymbol{w}]$. By the properties of eigenpairs we get $\boldsymbol{G}\boldsymbol{W} = \mu\boldsymbol{W}$. Applying a consistent matrix norm to this gives us

$$|\mu|||\boldsymbol{W}|| = ||\mu\boldsymbol{W}|| = ||\boldsymbol{G}\boldsymbol{W}|| \leq ||\boldsymbol{G}||||\boldsymbol{W}||.$$

As we know that $||\boldsymbol{W}|| \neq 0$ we must have that $|\mu| \leq ||\boldsymbol{G}||$. As this must hold for all μ of $\boldsymbol{G}$ and the spectral radius is simply the maximal eigenvalue

we now know that $\rho(\boldsymbol{G}) \leq ||\boldsymbol{G}||$.

Now all that is left to show is that if $\rho(\boldsymbol{G}) < 1$ then $||\boldsymbol{G}|| < 1$. This can be shown by proving that $\rho(\boldsymbol{G}) \leq ||\boldsymbol{G}|| \leq \rho(\boldsymbol{G}) + \varepsilon$ where $\varepsilon > 0$. We have already shown the first inequality so what remains is the show that the second inequality also holds. To do this we need the Schur Triangulation Theorem, also known as Schur Decomposition (see eg Theorem 5.13 in Lyche (2013)), which states that for each $\boldsymbol{A} \in \mathbb{C}^{n\times n}$ there exists a unitary matrix $\boldsymbol{U} \in \mathbb{C}^{n\times n}$such that $\boldsymbol{R} = \boldsymbol{U}^*\boldsymbol{A}\boldsymbol{U}$ is upper triangular.

We denote the eigenvalues of $\boldsymbol{G}$ by $\mu_1, \ldots, \mu_n$. From Schurs Triangulation theorem we know that there exists an upper triangular matrix $\boldsymbol{R} = [r_{ij}]$ such that $\boldsymbol{U}^*\boldsymbol{G}\boldsymbol{U} = \boldsymbol{R}$. We now define $\boldsymbol{D}_t = diag(t, t^2, \ldots, t^n) \in \mathbb{R}^{n\times n}$ for $t > 0$. We now combine $\boldsymbol{R}$ and $\boldsymbol{D}$ into the matrix $\boldsymbol{D}_t\boldsymbol{R}\boldsymbol{D}_t^{-1}$ where we see that the (i, j) element in this matrix is given by $t^{i-j}r_{ij}$ for all i, j.

We now introduce a new norm defined such that for each $\boldsymbol{B} \in \mathbb{C}^{n\times n}$ and $t > 0$ we have $||\boldsymbol{B}||_t = ||\boldsymbol{D}_t\boldsymbol{U}^*\boldsymbol{B}\boldsymbol{U}\boldsymbol{D}_t^{-1}||_1$. It can be shown that this is a consistent norm. Therefore all previous results also holds for this norm.

We choose a value of t that is so large that all the off-diagonal elements of $\boldsymbol{D}_t\boldsymbol{R}\boldsymbol{D}_t^{-1}$ are less than ε. By defining $||\boldsymbol{B}|| = ||\boldsymbol{B}||_t$ we now get

$$\begin{aligned}
||\boldsymbol{G}|| &= ||\boldsymbol{D}_t\boldsymbol{U}^*\boldsymbol{G}\boldsymbol{U}\boldsymbol{D}_t^{-1}||_1 \\
&= ||\boldsymbol{D}_t\boldsymbol{R}\boldsymbol{D}_t^{-1}||_1 \\
&= \max_{1\leq j\leq n} \sum_{i=1}^{n} |(\boldsymbol{D}_t\boldsymbol{R}\boldsymbol{D}_t^{-1})_{ij}| \\
&\leq \max_{1\leq j\leq n} (|\mu_j| + \varepsilon) \\
&= \rho(\boldsymbol{G}) + \varepsilon.
\end{aligned}$$

As we have now shown that both parts of the inequality $\rho(\boldsymbol{G}) \leq ||\boldsymbol{G}|| \leq \rho(\boldsymbol{G}) + \varepsilon$ holds we have shown that $\rho(\boldsymbol{G}) < 1$ is both a necessary and sufficient condition for $\lim_{k\to\infty} \boldsymbol{G}^k = 0$ and thus we have completed the proof needed to show

$$\lim_{k\to\infty} \boldsymbol{G}^k = 0 \Leftrightarrow \rho(\boldsymbol{G}) < 1.$$

4.6.2 Convergence when $\boldsymbol{A}$ is Strictly or Irreducibly Diagonally Dominant

In both Section 4.4.1 and Section 4.5.1 we have mentioned that these methods converge if the matrix $\boldsymbol{A}$ is strictly or irreducibly diagonally dominant.

We will now show why this guarantees convergence. This proof follows Section 4.2.3 in Saad (2003). As we now need to use the properties of the specific iteration matrix for each method we have to show this separately. The arguments are very similar, but not identical.

We start by showing it for Jacobi's method.
If $\boldsymbol{A}$ is strictly diagonally dominant we know that

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \Leftrightarrow \sum_{j \neq i} \frac{|a_{ij}|}{|a_{ii}|} < 1.$$

We know from the discussion earlier in this section that for Jacobi's method to converge we must have $||\boldsymbol{G}|| < 1$ for a consistent norm. We know the max-norm is consistent so we will use this to show convergence. Because of the way $\boldsymbol{G}$ is constructed in Jacobi's method and because $\boldsymbol{A}$ is strictly diagonally dominant, we get:

$$\begin{aligned} ||\boldsymbol{G}||_\infty &= ||\boldsymbol{D}^{-1}\boldsymbol{R}||_\infty \\ &= \max_{1 \leq i \leq m} \sum_{j \neq i} \frac{|r_{ij}|}{|d_{ii}|} \\ &= \max_{1 \leq i \leq m} \sum_{j \neq i} \frac{|a_{ij}|}{|a_{ii}|} \\ &< 1. \end{aligned}$$

As previously discussed we know that when $||\boldsymbol{G}|| < 1$ we are guaranteed convergence so we see from this that Jacobi's method always will converge if $\boldsymbol{A}$ is strictly diagonally dominant.

We want to show that the same is true for Gauss-Seidel's method. We let μ be an eigenvalue of the iteration matrix $\boldsymbol{G} = (\boldsymbol{D} + \boldsymbol{L})^{-1}\boldsymbol{U}$ and let $\boldsymbol{w}$ be the associated eigenvector. We scale this vector so that $|w_m| = 1$ and $|w_i| < 1$ for $i \neq m$ where m is the index of the component of $\boldsymbol{w}$ with the

largest absolute value.

$$\begin{aligned}
\boldsymbol{G}\boldsymbol{w} &= \mu\boldsymbol{w} \\
(\boldsymbol{D}+\boldsymbol{L})^{-1}\boldsymbol{U}\boldsymbol{w} &= \mu\boldsymbol{w} \\
\boldsymbol{U}\boldsymbol{w} &= \mu(\boldsymbol{D}+\boldsymbol{L})\boldsymbol{w} \\
\sum_{j<m} a_{mj}w_j &= \mu(a_{mm}w_m + \sum_{j>m} a_{mj}w_j) \\
\mu &= \frac{\sum_{j<m} a_{mj}w_j}{a_{mm}w_m + \sum_{j>m} a_{mj}w_j} \\
|\mu| &= \left|\frac{\sum_{j<m} a_{mj}w_j}{a_{mm}w_m + \sum_{j>m} a_{mj}w_j}\right| \\
&\leq \frac{|\sum_{j<m} a_{mj}w_j|}{|a_{mm}w_m + \sum_{j>m} a_{mj}w_j|} \\
&\leq \frac{\sum_{j<m} |a_{mj}||w_j|}{|a_{mm}||w_m| + \sum_{j>m} |a_{mj}||w_j|} \\
&\leq \frac{\sum_{j<m} |a_{mj}|}{|a_{mm}| + \sum_{j>m} |a_{mj}|}.
\end{aligned}$$

For simplicity we define the notation

$$\begin{aligned}
d &= |a_{mm}|, \\
\sigma_1 &= \sum_{j>m} |a_{mj}|, \\
\sigma_2 &= \sum_{j<m} |a_{mj}|.
\end{aligned}$$

d, σ_1 and σ_2 are obviously non-negative and as $\boldsymbol{A}$ is strictly diagonally dominant in this case we know that $d - \sigma_1 - \sigma_2 > 0$

$$\begin{aligned}
|\mu| &\leq \frac{\sigma_2}{d-\sigma_1} \\
&= \frac{\sigma_2}{d-\sigma_1+\sigma_2-\sigma_2} \\
&= \frac{\sigma_2}{\sigma_2+(d-\sigma_1-\sigma_2)} \\
&< 1.
\end{aligned}$$

As this must hold for all the eigenvalues of $\boldsymbol{G}$ we know that $\rho(\boldsymbol{G}) < 1$ and as previously shown this means the method converges. So as long as $\boldsymbol{A}$ is strictly diagonally dominant we know that Gauss-Seidel's method converges.

What we also mentioned in Section 4.4.1 and Section 4.5.1, but have not

shown, is that these methods converges if $\boldsymbol{A}$ is irreducibly diagonally dominant. To prove that this also holds we start the same way as we have for strict diagonal dominance, but both Gauss-Seidel's method and Jacobi's method will now give us that $|\mu| \leq 1$ instead of $|\mu| < 1$. So in addition we need to show that $|\mu|$ can not be equal to 1. We do this by using a quick proof by contradiction to prove that $\rho(\boldsymbol{G}) < 1$. We start by assuming that μ is an eigenvalue of $\boldsymbol{G}$ and that $|\mu| = 1$. For this proof we need to express $\boldsymbol{G}$ a bit differently. We know that for both our methods this iteration matrix consists of two parts. One matrix that needs to be inverted and then multiplied with another matrix. We can write $\boldsymbol{G} = \boldsymbol{M}^{-1}\boldsymbol{N}$. We know, by definition, that $\boldsymbol{G} - \mu\boldsymbol{I}$ is singular, and because of this we can define a matrix $\boldsymbol{A}' = \boldsymbol{N} - \mu\boldsymbol{M}$ which would also be singular. When $|\mu| = 1$ it is obvious that $\boldsymbol{A}'$ is irreducibly diagonally dominant. From Corollary 4.8 in Saad (2003) we know that when a matrix is strictly or irreducibly diagonally dominant then it must be non-singular. We therefore get a contradiction here as $\boldsymbol{A}'$ can not both be non-singular and singular at the same time. From this we know that $|\mu|$ can not be equal to 1 and we must have that $|\mu| < 1$. Thereby we have shown that the methods converge when the matrices are irreducibly diagonally dominant as well.

4.6.3 Convergence for Gauss-Seidel's Method when $\boldsymbol{A}$ is Symmetric Positive Definite

Gauss-Seidel's method has the property that it converges if $\boldsymbol{A}$ is symmetric positive definite. As this is a very useful property we want to show why this is true. The proof follows Shönlieb (2013).

When $\boldsymbol{A}$ is symmetric positive definite we know that $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{L} + \boldsymbol{U} = \boldsymbol{D} + \boldsymbol{U}^T + \boldsymbol{U}$. We know that $\boldsymbol{D}$ is symmetric positive definite. As it is diagonal, it is obviously symmetric and as $\boldsymbol{A}$ is positive definite and $\boldsymbol{D}$ contains the diagonal elements of $\boldsymbol{A}$, $\boldsymbol{D}$ must also be positive definite.

For Gauss-Seidel's method when $\boldsymbol{A}$ is symmetric positive definite we get the iteration matrix

$$\begin{aligned}\boldsymbol{G} &= -(\boldsymbol{D} + \boldsymbol{L})^{-1}\boldsymbol{U} \\ &= -(\boldsymbol{D} + \boldsymbol{U}^T)^{-1}\boldsymbol{U} \\ &= -(\boldsymbol{A} - \boldsymbol{U}^T - \boldsymbol{U} + \boldsymbol{U}^T)^{-1}\boldsymbol{U} \\ &= -(\boldsymbol{A} - \boldsymbol{U})^{-1}\boldsymbol{U}.\end{aligned}$$

We have $\boldsymbol{G}\boldsymbol{w} = \mu\boldsymbol{w}$ where μ is an eigenvalue and $\boldsymbol{w}$ is an eigenvector of $\boldsymbol{G}$. Because $\boldsymbol{A}$ is non-singular we know that it's eigenvalues are non-zero and because of the way $\boldsymbol{G}$ is defined from $\boldsymbol{A}$ we know that the eigenvalues μ of

$\boldsymbol{G}$ are unequal to 1.

$$\begin{aligned}
\boldsymbol{G}\boldsymbol{w} &= \mu\boldsymbol{w} \\
-(\boldsymbol{A}-\boldsymbol{U})^{-1}\boldsymbol{U}\boldsymbol{w} &= \mu\boldsymbol{w} \\
-\boldsymbol{U}\boldsymbol{w} &= \mu(\boldsymbol{A}-\boldsymbol{U})\boldsymbol{w} \\
\mu\boldsymbol{U}\boldsymbol{w}-\boldsymbol{U}\boldsymbol{w} &= \mu\boldsymbol{A}\boldsymbol{w} \\
(\mu-1)\boldsymbol{U}\boldsymbol{w} &= \mu\boldsymbol{A}\boldsymbol{w} \\
\boldsymbol{U}\boldsymbol{w} &= \frac{\mu}{\mu-1}\boldsymbol{A}\boldsymbol{w}.
\end{aligned}$$

We now multiply by the transpose of $\boldsymbol{w}$, but as we might have complex values in the eigenvector we also have to complex conjugate it. We denote this complex conjugate transpose as $\bar{\boldsymbol{w}}^T = \boldsymbol{w}^*$.

$$\boldsymbol{w}^*\boldsymbol{U}\boldsymbol{w} = \frac{\mu}{\mu-1}\boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w}.$$

We can write $\boldsymbol{w} = \boldsymbol{u} + i\boldsymbol{v}$ where both $\boldsymbol{u}$ and $\boldsymbol{v}$ are real. If $\boldsymbol{w}$ only have real elements then $\boldsymbol{v} = 0$ and the same arguments will still hold.

We get

$$\boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w} = \boldsymbol{u}^T\boldsymbol{A}\boldsymbol{u} + \boldsymbol{v}^T\boldsymbol{A}\boldsymbol{v},$$

and from this and the fact that $\boldsymbol{A}$ is symmetric positive definite we see that $\boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w} > 0$. Using the same argument we also get that $\boldsymbol{w}^*\boldsymbol{D}\boldsymbol{w} = \boldsymbol{w}^*(\boldsymbol{A}-\boldsymbol{U}^T-\boldsymbol{U})\boldsymbol{w} > 0$.

We can use this latest inequality to obtain properties for the eigenvalues.

$$\begin{aligned}
0 < \boldsymbol{w}^*(\boldsymbol{A}-\boldsymbol{U}^T-\boldsymbol{U})\boldsymbol{w} &= \boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w} - \boldsymbol{w}^*\boldsymbol{U}^T\boldsymbol{w} - \boldsymbol{w}^*\boldsymbol{U}\boldsymbol{w} \\
&= \boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w} - (\boldsymbol{w}^*\boldsymbol{U}\boldsymbol{w})^* - \boldsymbol{w}^*\boldsymbol{U}\boldsymbol{w} \\
&= \boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w} - \left(\frac{\mu}{\mu-1}\boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w}\right)^* - \frac{\mu}{\mu-1}\boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w} \\
&= \boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w} - \frac{\bar{\mu}}{\bar{\mu}-1}\boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w} - \frac{\mu}{\mu-1}\boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w} \\
&= \left(1 - \frac{\bar{\mu}}{\bar{\mu}-1} - \frac{\mu}{\mu-1}\right)\boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w} \\
&= \left(\frac{(\bar{\mu}-1)(\mu-1) - \bar{\mu}(\mu-1) - \mu(\bar{\mu}-1}{(\bar{\mu}-1)(\mu-1)}\right)\boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w} \\
&= \left(\frac{\mu\bar{\mu} - \mu - \bar{\mu} - 1 - \mu\bar{\mu} + \bar{\mu} - \mu\bar{\mu} + \mu}{(\mu-1)\overline{\mu-1}}\right)\boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w} \\
&= \left(\frac{|\mu|^2 - 2|\mu|^2 + 1}{|\mu-1|^2}\right)\boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w} \\
&= \frac{1-|\mu|^2}{|\mu-1|^2}\boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w}.
\end{aligned}$$

We know that $\mu \neq 1$ so we must have $|\mu - 1|^2 > 0$ and as $\boldsymbol{w}^*\boldsymbol{A}\boldsymbol{w} > 0$ we know that $1 - |\mu|^2$ must also be positive at all times. So we know that $|\mu| < 1$ for all μ and then by definition we must have $\rho(\boldsymbol{G}) < 1$ and, as we have shown in Section 4.6.1, this guarantees convergence so Gauss-Seidel's method will always converge when $\boldsymbol{A}$ is symmetric positive definite.

4.7 More on the Convergence of Richardson's Method

As shown in Section 4.3.1 it is easy to obtain the necessary conditions to guarantee convergence for Richardson's method. The results derived in Section 4.6.1 also applies to Richardson's method as these results are derived for any iterative method that can be written on the form $\boldsymbol{x}^{(k+1)} = \boldsymbol{G}\boldsymbol{x}^{(k)} + \boldsymbol{c}$. From Section 4.3 we know that the matrix form of Richardson's method is

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \alpha(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)}).$$

This can be rewritten as

$$\begin{aligned}\boldsymbol{x}^{(k+1)} &= \boldsymbol{x}^{(k)} + \alpha\boldsymbol{b} - \alpha\boldsymbol{A}\boldsymbol{x}^{(k)} \\ \boldsymbol{x}^{(k+1)} &= (\boldsymbol{I} - \alpha\boldsymbol{A})\boldsymbol{x}^{(k)} + \alpha\boldsymbol{b}.\end{aligned}$$

We see that by setting

$$\boldsymbol{G} = \boldsymbol{I} - \alpha\boldsymbol{A} \qquad \text{and} \qquad \boldsymbol{c} = \alpha\boldsymbol{b},$$

we can write Richardson's method on the same general form as we used in the previous section. We also see that the convergence requirement $||\boldsymbol{I} - \alpha\boldsymbol{A}|| < 1$ we got in Section 4.3.1 fits with the results we derived in Section 4.6.1.

As $\boldsymbol{G} = \boldsymbol{I} - \alpha\boldsymbol{A}$ and α is a parameter we get that the matrix $\boldsymbol{G}$ depends on this parameter, so for Richardson's method we write $\boldsymbol{G}(\alpha)$ instead of $\boldsymbol{G}$.

As we see from $||\boldsymbol{I} - \alpha\boldsymbol{A}|| < 1$ the convergence of Richardson's method might depend heavily on choosing the right value of α. We will now take a closer look at this parameter and see if we can find some properties for α that when fulfilled guarantees convergence. This proof follows Section 8.3.1 in Lyche (2013)

We make the assumption that $\boldsymbol{A}$ only has positive eigenvalues, and start by looking at the eigenvalues of $\boldsymbol{G}(\alpha)$. These are $\mu_j(\alpha) = 1 - \alpha\lambda_j$ for $j = 1, \ldots, n$ where λ_j are the eigenvalues of $\boldsymbol{A}$. As previously discussed we must have $\rho(\boldsymbol{G}(\alpha)) < 1$ which means that we need to have $\max_{1 \leq j \leq n} |\mu_j(\alpha)| < 1$. To ensure this we can limit the value for α. We see that $\max_j \mu_j < 1$ if

and only if $\alpha > 0$. To fulfil the convergence requirement we must also have $\min_j \mu_j > -1$.

$$\begin{aligned}
\min_j \mu_j &= 1 - \alpha \max_j |\lambda_j| \\
&= 1 - \alpha\rho(\boldsymbol{A}) \\
-1 &< 1 - \alpha\rho(\boldsymbol{A}) \\
\alpha\rho(\boldsymbol{A}) &< 2 \\
\alpha &< \frac{2}{\rho(\boldsymbol{A})}.
\end{aligned}$$

This gives that $\rho(\boldsymbol{G}(\alpha)) < 1$ when $0 < \alpha < \frac{2}{\rho(A)}$. So long as α is in this interval the method will converge, given that $\boldsymbol{A}$ has only positive eigenvalues.

We will now show that $\alpha^* = \frac{2}{\lambda_{max}+\lambda_{min}}$ is the optimal value of α and that $\min_\alpha \boldsymbol{G}(\alpha) = \boldsymbol{G}(\alpha^*)$. This proof is also based on Section 8.3.1 in Lyche (2013). We start with some simple calculation

$$\begin{aligned}
1 - \alpha^*\lambda_{min} &= 1 - \frac{2\lambda_{min}}{\lambda_{max} + \lambda_{min}} \\
&= \frac{\lambda_{max} + \lambda_{min} - 2\lambda_{min}}{\lambda_{max} + \lambda_{min}} \\
&= \frac{\lambda_{max} - \lambda_{min}}{\lambda_{max} + \lambda_{min}} \\
&= \frac{\lambda_{max} - \lambda_{min} + \lambda_{max} - \lambda_{max}}{\lambda_{max} + \lambda_{min}} \\
&= \frac{2\lambda_{max}}{\lambda_{max} + \lambda_{min}} - \frac{\lambda_{max} + \lambda_{min}}{\lambda_{max} + \lambda_{max}} \\
&= \alpha^*\lambda_{max} - 1.
\end{aligned}$$

Because $1 - \alpha^*\lambda_{min} = \alpha^*\lambda_{max} - 1$ we have

$$\begin{aligned}
\rho(\boldsymbol{G}(\alpha^*)) &= 1 - \alpha^*\lambda_{min} \\
&= \frac{\lambda_{max} - \lambda_{min}}{\lambda_{max} + \lambda_{min}} \\
&= \frac{\frac{\lambda_{max}}{\lambda_{min}} - \frac{\lambda_{min}}{\lambda_{min}}}{\frac{\lambda_{max}}{\lambda_{min}} + \frac{\lambda_{min}}{\lambda_{min}}} \\
&= \frac{\kappa - 1}{\kappa + 1},
\end{aligned}$$

where $\kappa = \frac{\lambda_{max}}{\lambda_{min}}$.

If $0 \leq \alpha < \alpha^*$ we now know that

$$\rho(\boldsymbol{G}(\alpha)) \geq 1 - \alpha\lambda_{min} > 1 - \alpha^*\lambda_{min} = \rho(\boldsymbol{G}(\alpha^*)),$$

and if $\alpha^* < \alpha \leq \frac{2}{\rho(\boldsymbol{A})}$ that

$$-\rho(\boldsymbol{G}(\alpha)) \leq 1 - \alpha\lambda_{max} < 1 - \alpha^*\lambda_{max} = -\rho(\boldsymbol{G}(\alpha^*)).$$

From this we see that $\rho(\boldsymbol{G}(\alpha)) > \rho(\boldsymbol{G}(\alpha^*))$ so we must have $\boldsymbol{G}(\alpha) > \boldsymbol{G}(\alpha^*)$ and therefore

$$\min_{\alpha} \boldsymbol{G}(\alpha) = \boldsymbol{G}(\alpha^*).$$

$\alpha^* = \frac{2}{\lambda_{max}+\lambda_{min}}$ is the optimal choice for α as it minimizes all $|1 - \alpha\lambda_j|$ and therefore gives the smallest errors and the fastest convergence.

So we now know that when $\boldsymbol{A}$ has only positive eigenvalues that Richardson's method converges when $0 < \alpha < \frac{2}{\rho(\boldsymbol{A})}$ and that the optimal value of α is $\alpha^* = \frac{2}{\lambda_{max}+\lambda_{min}}$.

If however $\boldsymbol{A}$ has both positive and negative eigenvalues then the method diverges for any α when the initial error $\boldsymbol{e}^{(0)}$ has nonzero components in the eigenvectors corresponding to the negative eigenvalues.

Rate of Convergence for Richardson's method

Using what we have just shown regarding the criteria for convergence and the optimal value of α we can find the rate of convergence for this method. The disadvantage of this result is that it only holds when $\boldsymbol{A}$ is symmetric positive definite, as we take advantage of the fact that for symmetric positive definite matrices the spectral norm $||\cdot||_2$ is equal to the spectral radius. As the spectral norm is consistent we can also use the results in previous sections

When using $\alpha = \alpha^*$ we get

$$\begin{aligned}
||\boldsymbol{x}^{(k)} - \boldsymbol{x}||_2 &\leq ||\boldsymbol{G}(\alpha^*)||_2^k ||\boldsymbol{x}^{(0)} - \boldsymbol{x}||_2 \\
||\boldsymbol{x}^{(k)} - \boldsymbol{x}||_2 &\leq \max_j |\mu_j(\alpha^*)|^k ||\boldsymbol{x}^{(0)} - \boldsymbol{x}||_2 \\
||\boldsymbol{x}^{(k)} - \boldsymbol{x}||_2 &\leq \rho(\boldsymbol{G}(\alpha^*))^k ||\boldsymbol{x}^{(0)} - \boldsymbol{x}||_2 \\
||\boldsymbol{x}^{(k)} - \boldsymbol{x}||_2 &\leq \left(\frac{\kappa - 1}{\kappa + 1}\right)^k ||\boldsymbol{x}^{(0)} - \boldsymbol{x}||_2.
\end{aligned}$$

4.8 Number of Operations

The number of operations required to solve each of the methods described in this chapter will largely be governed by the properties of $\boldsymbol{A}$. For each method the number of operations required to solve the matrix product $\boldsymbol{G}\boldsymbol{x}^{(k)}$ will be the governing factor.

For the matrix methods we get this matrix product directly and as such the number of operations required will be $\mathcal{O}(n^2)$ as this is the number of operations required to solve a matrix product. The rest of the method will only contribute various $\mathcal{O}(n)$ and as n increases this contribution will be rapidly outdistanced by the $\mathcal{O}(n^2)$ factor from the matrix product. This matrix product has to be computed for each iteration through the method so in addition to the $\mathcal{O}(n^2)$ we get the number of iterations k as well so in fact the number of iterations for each matrix form will be $\mathcal{O}(kn^2)$, but as k generally will be much smaller than the number of unknowns n the matrix forms can be viewed as $\mathcal{O}(n^2)$.

The component form of the various methods will also require $\mathcal{O}(n^2)$. This is because they loop over the n equations and for each equation they compute a sum of n factors. They also depend on the number of iterations required for convergence so in general they will be $\mathcal{O}(kn^2)$, but as long as $k << n$ they are $\mathcal{O}(n^2)$.

There are cases where the number of operations will be significantly lower than the scenario briefly discussed above. These cases occur when $\boldsymbol{A}$ is sparse. The more zeroes $\boldsymbol{A}$ contains the more advantageous it becomes to use an iterative method instead of the more computationally expensive alternatives like Gaussian elimination.

For the matrix forms the matrix product requires $\mathcal{O}(n)$ operations when $\boldsymbol{A}$ is sparse instead of $\mathcal{O}(n^2)$ when it is full. So in this case the number of iterations required will be $\mathcal{O}(kn)$, or $\mathcal{O}(n)$ when $k << n$, which is significantly better than the general case.

The same is true for the component forms. While we still need to loop through the n equations, the number of factors we get in the sum will be significantly lower than n. If we for instance are solving a partial differential equation with a finite difference scheme we will get a matrix with just a few bands of non-zero elements. So in the three-dimensional case we will get three bands of ones on each side of the diagonal so the sum will contain 7 elements instead of n elements for each equation. Then the method will require $\mathcal{O}(7n)$ for each k instead of $\mathcal{O}(n^2)$. And generally the number of non-zero elements in each sum will be so much smaller than the number of unknowns so that we can simply view it as requiring $\mathcal{O}(n)$ operations. Again we also depend on the number of iterations k so we get $\mathcal{O}(kn)$ operations or $\mathcal{O}(n)$ operations depending on how much larger n is than k.

Taking the fact that $\boldsymbol{A}$ is sparse into account when discussing the theoretical number of operations for each method is rather easy. It becomes more difficult to take advantage of this in practice when programming solvers for

these iterative methods. If the programming language we use do not support taking advantage of sparse structures and we do not write our program to take advantage of this structure, then the number of operations will be $\mathcal{O}(n^2)$ regardless of the structure of $\boldsymbol{A}$. When programming in MATLAB the program will take advantage of this "behind the scenes" and we can write the code without thinking of whether $\boldsymbol{A}$ is sparse or not. The only thing we have to remember is to store it as a sparse matrix and MATLAB will do the rest. When coding in languages that do not have this support we have to take it into account when writing the solver itself, to gain fewer operations and get a shorter runtime for the program. This requires us to know the structure of the matrix we will be working on and makes it much more difficult to create general code that can be used to solve any linear system $\boldsymbol{Ax} = \boldsymbol{b}$.

4.9 Conjugate Gradient Method

The Conjugate Gradient method is not included in the classical iterative methods. Its main use is for iteratively solving large sparse linear systems so its purpose is the same as the classical iterative methods that we have discussed before. The advantage of the Conjugate Gradient method is that it is much faster than these methods. The disadvantage is that for this method to work we must have a symmetric positive definite system, whereas the classical iterative methods work regardless of the properties of the matrix, as long as the main diagonal is non-zero.

The Conjugate-Gradient method:
We choose a starting vector $\boldsymbol{x}_0 \in \mathbb{R}^n$ and start with defining $\boldsymbol{p}_0 = \boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$. If $\boldsymbol{r}_0 = \boldsymbol{0}$ then $\boldsymbol{x}_0$ is the exact solution and so we are finished, otherwise we start iterating through the conjugate gradient method as follows

For k = 0,1,2,...

$$\begin{aligned}
\boldsymbol{t}_k &= \boldsymbol{A}\boldsymbol{p}_k, \\
\alpha_k &= \frac{\boldsymbol{r}_k^T \boldsymbol{r}_k}{\boldsymbol{p}_k^T \boldsymbol{t}_k}, \\
\boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k, \\
\boldsymbol{r}_{k+1} &= \boldsymbol{r}_k - \alpha_k \boldsymbol{t}_k, \\
\beta_k &= \frac{\boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1}}{\boldsymbol{r}_k^T \boldsymbol{r}_k}, \\
\boldsymbol{p}_{k+1} &= \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k.
\end{aligned}$$

The conjugate gradient method is a direct method. We know the $n+1$ residuals $\boldsymbol{r}_0, \ldots, \boldsymbol{r}_n$ cannot all be non-zero as $\dim \mathbb{R}^n = n$. Because of this,

and because the residuals are orthogonal, we know that will find the exact solution in at most n iterations.

4.9.1 Convergence

To discuss the convergence of the Conjugate Gradient method we need to introduce a new inner product, called the $\boldsymbol{A}$-inner product, and the corresponding norm, called the $\boldsymbol{A}$-norm. These are defined by

$$\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{y} \quad ||\boldsymbol{x}||_{\boldsymbol{A}} = \sqrt{\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x}} \quad \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n.$$

We need that $\boldsymbol{A}$ is symmetric positive definite for this inner product to actually be an inner product. The Conjugate Gradient method only works for symmetric positive definite matrices, so we know that this will always be the case when using this inner product while discussing this method so the inner product will work for the purposes we need it for. Using this new $\boldsymbol{A}$-norm we can find the upper bound for the error of the conjugate gradient method. This upper bound is

$$\frac{||\boldsymbol{x} - \boldsymbol{x}^{(k)}||_{\boldsymbol{A}}}{||\boldsymbol{x} - \boldsymbol{x}^{(0)}||_{\boldsymbol{A}}} \leq 2 \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^k < 2e^{-\frac{2}{\sqrt{\kappa}}k} \quad k \geq 0,$$

where $\kappa = \frac{\lambda_{max}}{\lambda_{min}}$ is the condition number of $\boldsymbol{A}$ and λ_{max} and λ_{min} are the largest and smallest eigenvalue of $\boldsymbol{A}$. As the proof for this is very complex we will not go through it here, but those interested in the full proof can consult Lyche (2013), Section 9.4.

Chapter 5

Serial Results

5.1 Introduction

We have implemented and solved the test problem described in Section 3.2 with the iterative methods described in Chapter 4 in MATLAB. The implementation of the iterative methods can be found in Appendix A. The code for generating the matrix $\boldsymbol{A}$ and solving the test problem can be found in Appendix B.

5.2 The Component Forms

When testing these implementations on our test problem we found that the component forms of the various methods where so much slower than their matrix equivalent that they would never be useful for our purposes. In this chapter we will therefore focus only on the results given by the matrix versions of the classical iterative methods. We found that the component forms used the exact same number of iterations and gave the exact same solution as their matrix equivalent. This is the expected behaviour as the component form is the exact same method as the corresponding matrix form. It is simply a different way to express the same method.

The reason for the large difference in time between the matrix and component forms in our tests is based on which programming language we are using. If we for instance used C or C++, there would be no difference in how we chose to express the method. We would be implementing the exact same thing as we would have to iterate over all the linear equations we have to solve. The only time it is relevant to make a distinction between the two forms is when programming in a language which support vectorization, like MATLAB. MATLAB has optimized routines for matrix-vector products which is using compiled code behind the scenes. This is much faster than

using `for`-loops which would only run unoptimized, uncompiled MATLAB code.

When programming in MATLAB we can also take advantage of the fact that the matrix for our test problem is sparse. Using matrix-vector products when implementing the iterative methods allows us to take full advantage of this. Using `for`-loops will not give us the same advantage as we might end up iterating over a lot more elements than we need to. As we discussed in Section 4.8 the number of operations for the matrix forms are $\mathcal{O}(n)$ when $\boldsymbol{A}$ is sparse, and we are able to take advantage of this fact. For the component forms we will be closer to $O(n^2)$ as MATLAB can not get the same advantage out of a `for`-loop. This is another reason for why the component implementation is slower than the matrix implementation when using MATLAB.

5.3 Results

The problem has been solved on a unit cube in space and for five seconds in time, meaning that we have iterated from $T_{start} = 0$ to $T_{end} = 5$ with the time step Δt. We have to use different values of Δt for the explicit and implicit methods based on the requirements to ensure stability and accuracy.

To ensure that the Explicit Euler method is stable this method uses the time step

$$\Delta t_E = \frac{\beta h^2}{6},$$

where $\beta = 100$

Through testing we found that the Crank-Nicolson scheme appears to be stable for $\Delta t < \frac{\beta h}{6}$. So we can use the same Δt values for both the implicit schemes as the primary constraint on Δt for the two implicit schemes is accuracy and not stability. As the implicit methods are not constrained by the stability requirements of Forward Euler we can run these methods with a much larger time step. Here we use

$$\Delta t = \frac{C\beta h}{6},$$

where $\beta = 100$ and $C < 1$. The only thing we have to consider when deciding the time step for the implicit methods is accuracy. We use different values of C and therefore of Δt for Backward Euler and Crank-Nicolson. If we discretize the derivative in time by using the Crank-Nicolson scheme we can get accurate results also for larger C, but $C = \frac{1}{4}$ is the largest where we see this method becoming more accurate than the explicit methods for the grids we are solving on. When using Backward Euler $C = \frac{1}{10}$ is the largest value for which we get acceptable accuracy for the iterative methods

for our grids. We could of course use a much smaller C to increase accuracy even more, but the trade off here is that the more time steps we take the longer the implicit methods take to solve the problem. So it is possible to get better accuracy, but it takes more time. And as Explicit Euler is so much faster to calculate for each time step we needed to find a value of C where we take as few time steps as possible, but still get an accurate enough result. $C = \frac{1}{4}$ seems to give the best results here when using Crank-Nicolson and $C = \frac{1}{10}$ gives the best results for Backward Euler. We will discuss both time usage and accuracy when using these C values in much more detail in the following sections.

The criteria for convergence for the iterative methods has here been that the residual must be smaller than the tolerance $\varepsilon = 10^{-7}$. Using a more relaxed criteria than this results in the methods having less than first order convergence towards the exact solution, and we want at least first order convergence for our methods. There is no reason for using a stricter tolerance in the iteration because of the error due to the discretization. When we iterate to a smaller tolerance than $\varepsilon = 10^{-7}$ the dominating part of the error comes from the discretization. Therefore we will not get a significantly more accurate solution even if we use a much smaller tolerance.

5.3.1 Number of unknowns

The number of unknowns for this problem is calculated by

$$n_x = n_y = n_z = \frac{1}{h} + 1,$$

$$n = n_x \cdot n_y \cdot n_z,$$

where n_i is the number of unknowns in direction i.

h	0.5	0.25	0.125	0.0625	0.03125	0.0156	0.0078
n	27	125	729	4913	35937	274625	2146689

Table 5.1: Number of unknowns for the grids

5.3.2 Eigenvalues

Richardson's methods requires the calculation of the maximum and minimum eigenvalues of the matrix $\boldsymbol{A}$. As this is a rather computationally expensive process, even in MATLAB, the eigenvalues have only been calculated up to the grid $h = 0.5^6$. In Table 5.2 we have also included the optimal α for Richardson's method, which is $\alpha = \frac{2}{\lambda_{max}+\lambda_{min}}$. Figure 5.1 shows a plot of the largest and smallest eigenvalues of $\boldsymbol{A}$.

n	27	125	729	4913	35937	274625
λ_{max}	0.48	1.92	7.68	30.72	122.88	491.52
λ_{min}	0.08	0.0937	0.0974	0.0984	0.0986	0.0987
α	3.5714	0.9932	0.2572	0.0649	0.0163	0.0041

Table 5.2: The largest and smallest eigenvalue of $\boldsymbol{A}$.

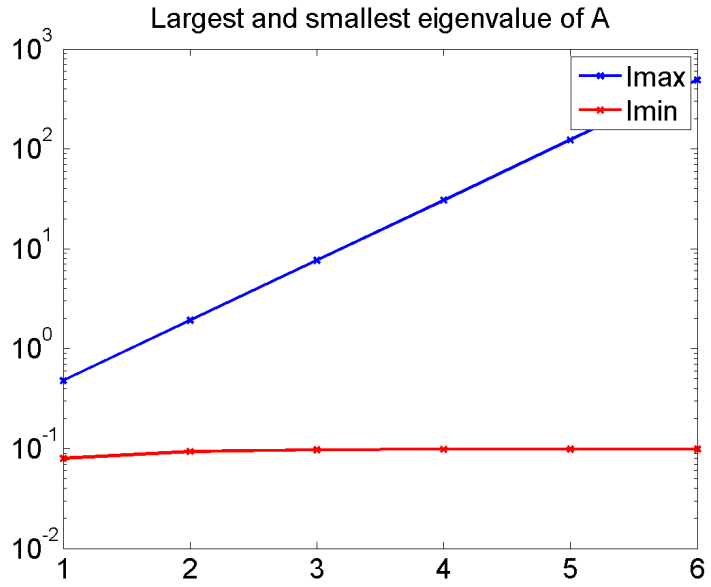


Figure 5.1: The largest and smallest eigenvalues of $\boldsymbol{A}$ for each grid. Relevant for the convergence rate of Richardson's method. The numbers along the x-axis are the exponents j in $h = 0.5^j$.

5.3.3 Initial Guess Vector

All our iterative methods require an initial guess vector $\boldsymbol{u}_0$. This initial guess vector is the starting point for the iterations at each time step. We found that using the result vector from the previous time step $\boldsymbol{u}^n$ as the initial guess vector for the following time step $\boldsymbol{u}_0^{n+1} = \boldsymbol{u}^n$, gave the best results when it came to rate of convergence.

Another property of using this vector is that the rate of convergence now depends on the time step Δt. The smaller the time step is the smaller the change in the solution vector $\boldsymbol{u}$ for each time step. Therefore we will need fewer and fewer iterations as the initial guess vector will get closer and closer to the solution at the current time step the smaller Δt becomes.

5.3.4 Convergence When Using Backward Euler in Time

The number of iterations required for each method is found in Table 5.3. Figure 5.2 shows the plots of some of these results. In this figure we have

not included plots of the iteration numbers for Richardson's method and the Conjugate-Gradient method. Results for Richardson's method are not included because the iteration numbers for some of the grids are so much larger than the other results, as we can see from Table 5.3. The results for the Conjugate-Gradient method has not been included as we have to solve this method with different boundary conditions than the other methods. We will get back to this method in Section 5.3.13.

Figure 5.3 shows the decrease in residual as we iterate using Jacobi's method for each of our grids. We use this residual to test if convergence has been reached for the classical iterative methods. The dotted black line is the tolerance for convergence, $\varepsilon = 10^{-7}$.

From Table 5.3 we see that Richardson's method requires a significantly larger number of iterations than the other methods before the error is sufficiently small. The reason for this is that as the grid is refined and $\boldsymbol{A}$ becomes larger, the largest eigenvalues of $\boldsymbol{A}$ also increases significantly as we see in Figure 5.1. We have implemented Richardson's method to use the optimal $\alpha = \frac{2}{\lambda_{max}+\lambda_{min}}$ to get the optimal convergence rate, as discussed in Section 4.7. Therefore a significant increase in eigenvalues will result in a significant decrease in α. Because of the way Richardson's method is constructed this will rapidly increase the number of iterations required for convergence as the steps taken to minimize the residual will be much smaller for each iteration and therefore we need many more iterations before the residual is small enough.

	0.5	0.5^2	0.5^3	0.5^4	0.5^5	0.5^6	0.5^7
k_R	5001	13	26	114	440	1629	-
k_J	8	1	9	16	27	46	80
k_{JW}	9	11	17	26	42	71	122
k_{GS}	6	6	8	11	18	30	51
k_{CG}	3	6	5	6	5	5	7
k_{BiCG}	1	1	1	2	2	3	6

Table 5.3: The number of iterations required for convergence for the different methods when using Backward Euler. The subscript denotes the methods. R, J and GS are Richardson's, Jacobi's and Gauss-Seidel's methods respectively. CG and BiCG are the Conjugate-Gradient method and the Biconjugate-Gradient method. JW is the weighted Jacobi method.

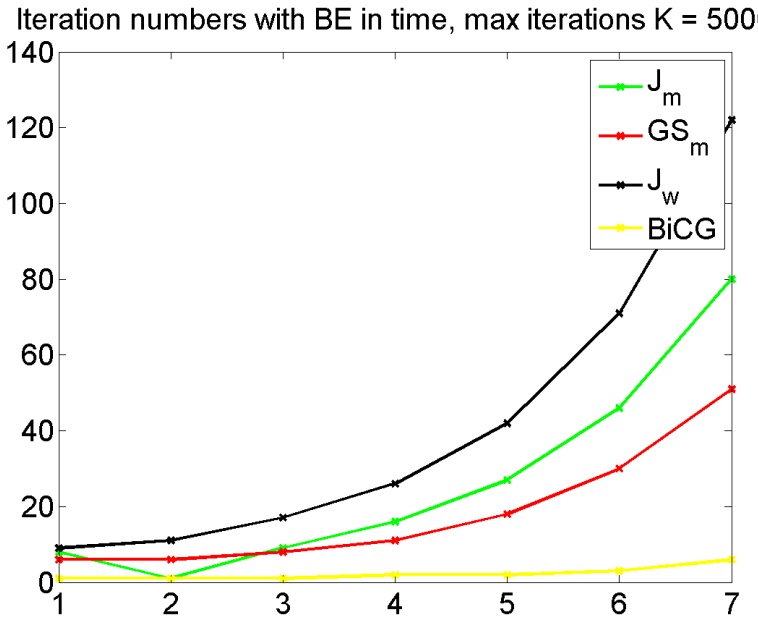


Figure 5.2: Plot of convergence rates for some methods when using Backward Euler (BE). The numbers along the x-axis are the exponents j in $h = 0.5^j$.

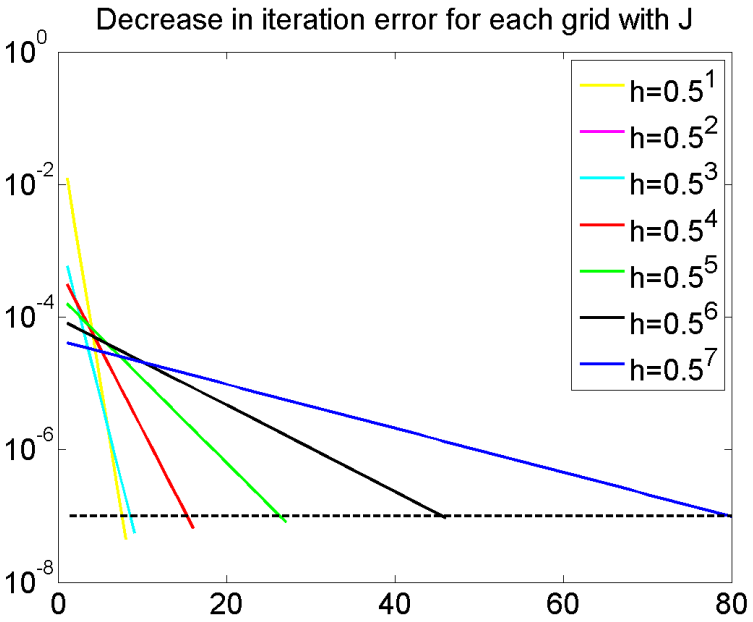


Figure 5.3: The decrease in residual as we iterate using Jacobi's method (J). The x-axis shows the number of iterations used.

The number of iterations required for the two other classical iterative methods is controlled by the norm of the iteration matrix. A plot of these norms for the four coarsest grids is included in Figure 5.4. As we discuss in Section

4.6 the error for Jacobi's and Gauss-Seidel's method is limited by

$$\begin{aligned} ||\boldsymbol{x}^{(k)} - \boldsymbol{x}|| &= ||\boldsymbol{G}^k(\boldsymbol{x}^{(0)} - \boldsymbol{x})|| \\ &\leq ||\boldsymbol{G}^k||||\boldsymbol{x}^{(0)} - \boldsymbol{x}|| \\ &\leq ||\boldsymbol{G}||^k||\boldsymbol{x}^{(0)} - \boldsymbol{x}||, \end{aligned}$$

where $\boldsymbol{G}$ is the iteration matrix for each method, $\boldsymbol{x}$ is the analytical solution and k is the number of iterations. As long as we use a norm which divides by the length of the vector, the initial error will be the same regardless of which of our grids we solve on. It is natural to use a norm which takes the length of the vector into account when we compare norms for different sized vectors. This means that the increase in iterations is related to only the behaviour of the norm $||\boldsymbol{G}||$.

When the iterative method has been allowed to iterate until convergence within a small tolerance, we can expect that the resulting error $||\boldsymbol{x}^{(k)} - \boldsymbol{x}||$ is approximately equal, regardless of which grid we have solved the problem on. Using this, we can derive an expression for how we can expect the norms to behave when solving our test problem. Here $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$ are the iteration matrices for each of the two grids and k_1 and k_2 are the number of iterations needed to reach convergence with the corresponding iteration matrix.

$$\begin{aligned} ||\boldsymbol{x}^{(k_1)} - \boldsymbol{x}|| &\approx ||\boldsymbol{x}^{(k_2)} - \boldsymbol{x}|| \\ ||\boldsymbol{G}_1||^{k_1}||\boldsymbol{x}^{(0)} - \boldsymbol{x}|| &\approx ||\boldsymbol{G}_2||^{k_2}||\boldsymbol{x}^{(0)} - \boldsymbol{x}|| \\ ||\boldsymbol{G}_1||^{k_1} &\approx ||\boldsymbol{G}_2||^{k_2} \\ k_1 \log||\boldsymbol{G}_1|| &\approx k_2 \log||\boldsymbol{G}_2|| \\ \log||\boldsymbol{G}_1|| &\approx \frac{k_2}{k_1} \log||\boldsymbol{G}_2|| \\ \log||\boldsymbol{G}_1|| &\approx \log||\boldsymbol{G}_2||^{\frac{k_2}{k_1}} \\ ||\boldsymbol{G}_1|| &\approx ||\boldsymbol{G}_2||^{\frac{k_2}{k_1}} \\ \sqrt[\frac{k_2}{k_1}]{||\boldsymbol{G}_1||} &\approx ||\boldsymbol{G}_2|| \\ ||\boldsymbol{G}_2|| &\approx \sqrt[\frac{k_2}{k_1}]{||\boldsymbol{G}_1||}. \end{aligned}$$

We are here looking at the relation of the error when solving the problem using an iterative methods on two different grids.

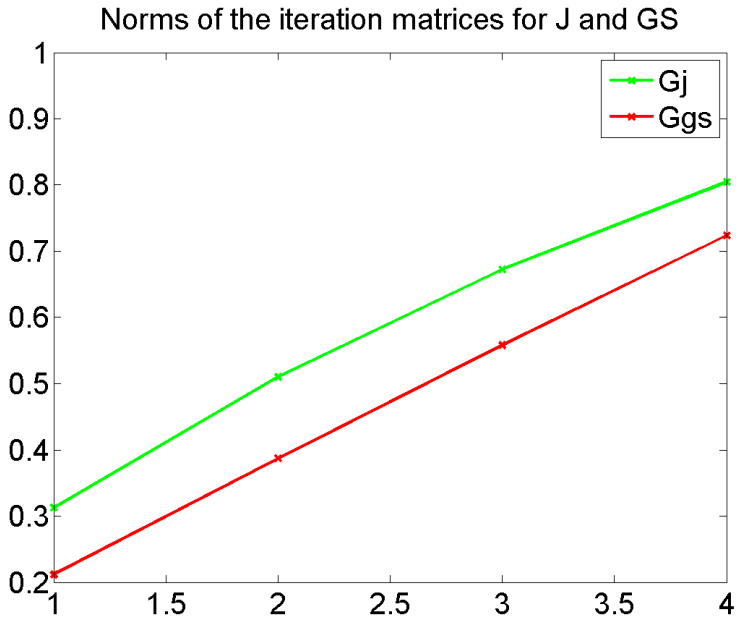


Figure 5.4: Norms of the iteration matrices for Jacobi's (J) and Gauss-Seidel's (GS) method for the four coarsest grids. Relevant for the convergence of these methods.

	0.5	0.5^2	0.5^3	0.5^4
$\|\|\boldsymbol{G}_J\|\|$	0.18246	0.2977	0.45462	0.62415
$\sqrt{\|\|\boldsymbol{G}_J\|\|}$	0.42715	0.54562	0.67425	0.79003
$\|\|\boldsymbol{G}_{GS}\|\|$	0.11842	0.2091	0.34388	0.51206
$\sqrt{\|\|\boldsymbol{G}_{GS}\|\|}$	0.34412	0.45728	0.58642	0.71558

Table 5.4: Norms and square root of the norms of the iteration matrices for Jacobi's (J) and Gauss-Seidel's (GS) method when using Backward Euler.

We see from Table 5.3 that both Jacobi's and Gauss-Seidel's method need a bit less than twice as many iterations to converge for each time the grid is refined. We know that the error for these methods are limited by $||\boldsymbol{G}||^k$. As we need close to twice as many iterations for each increase in the number of unknowns, we should have the relation

$$||\boldsymbol{G}_2|| \approx \sqrt{||\boldsymbol{G}_1||},$$

meaning that the norm of the current iteration matrix $\boldsymbol{G}$ should be smaller than the square root of norm of the previous iteration matrix. From Table 5.4 we see that the norms for the iteration matrices for our problem fits this relation. The norm of the next grid is even quite a bit smaller than the square root of the current grid. The iteration numbers we have match the theory for convergence of Jacobi's and Gauss-Seidel's method in Section 4.6.

The norms have only been calculated up to the grid $h = 0.5^4$. This is

because approximating norms is a very memory heavy operation in MATLAB, even when using spare matrices. We were not able to calculate the norms for the finer grids because we ran out of memory. As the number of iterations increase we know the norm increases towards 1, but as the methods converge for all calculated grids we know that it does not reach 1. We do also see from Table 5.4 that the increase in the norm becomes smaller and smaller so it is natural, from the data we have, to assume that the norms will approach, but never reach 1.

From Table 5.4 we see one of the reasons why Gauss-Seidel's method requires fewer iterations than Jacobi's method. Because the norm of the iteration matrix for Gauss-Seidel's method is smaller than the corresponding norm for Jacobi's method, the error for Gauss-Seidel's method will decrease faster and therefore this method requires fewer iterations before the criteria for convergence is reached.

5.3.5 Convergence When Using Crank-Nicolson in Time

The number of iterations required for each method is found in Table 5.5. Plots of the convergence rates for the various methods are in Figure 5.5. We have not included the results for Richardson's method and the Conjugate-Gradient method in this plot for the same reason as described in the previous section. Figure 5.6 shows the decrease in the residual as we iterate using Jacobi's method.

	0.5	0.5^2	0.5^3	0.5^4	0.5^5	0.5^6	0.5^7
k_R	5001	16	26	121	480	1822	-
k_J	8	1	10	19	35	63	113
k_{JW}	10	11	19	31	54	97	171
k_{GS}	6	7	8	12	23	40	70
k_{CG}	3	6	5	5	6	9	8
k_{BiCG}	1	1	1	2	4	5	7

Table 5.5: The number of iterations required for convergence for the different methods when using Crank-Nicolson. R, J and GS are Richardson's, Jacobi's and Gauss-Seidel's methods respectively. CG and BiCG are the Conjugate-Gradient method and the Biconjugate-Gradient method. JW is the weighted Jacobi method.

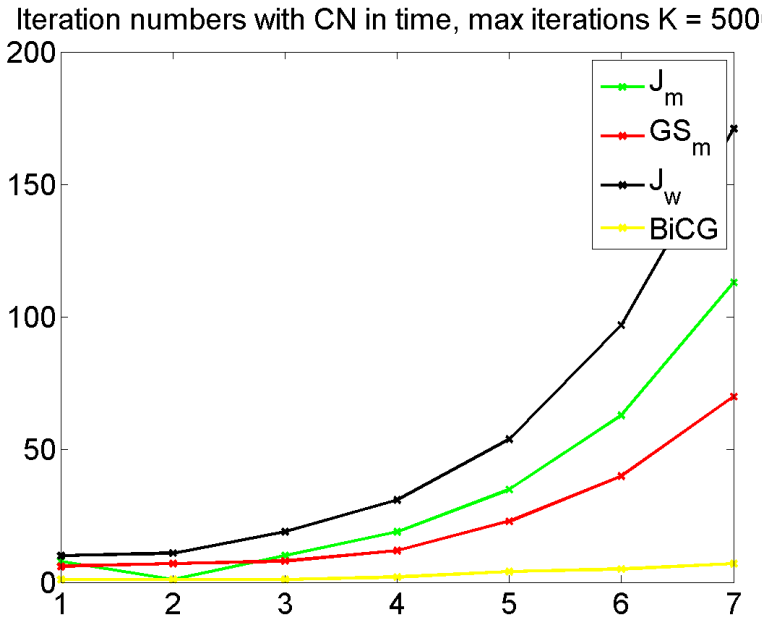


Figure 5.5: Number of iterations required for convergence when using Crank-Nicolson (CN). The numbers along the x-axis are the exponents of the grid parameter h.

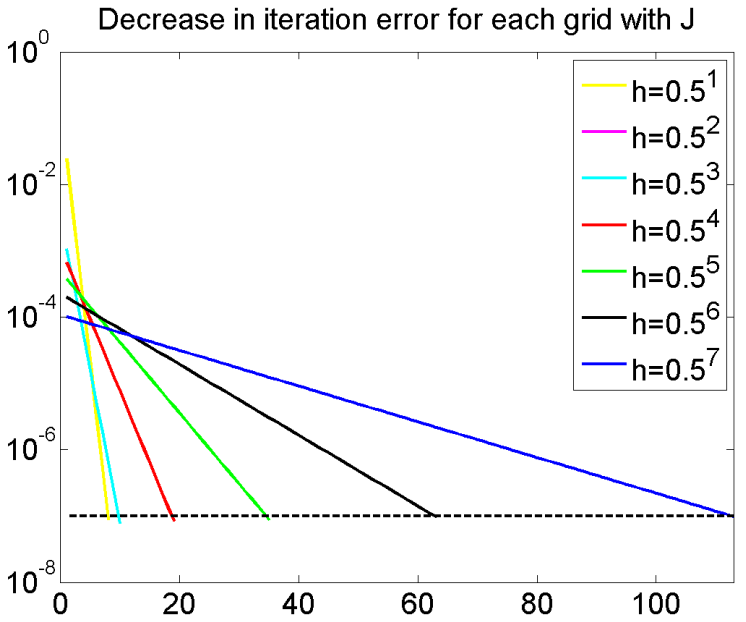


Figure 5.6: The decrease in residual as we iterate using Jacobi's method (J) with Crank-Nicolson. The x-axis shows the number of iterations used.

We see from the plot in Figure 5.5 and from Table 5.5 that the iterative methods require more iterations when we use Crank-Nicolson than for Backward Euler. There is a very simple explanation. To get accurate enough

results for Backward Euler we have to use a smaller Δt than what is required for Crank-Nicolson. When we take more time steps, like we do for Backward Euler, fewer iterations are required for each time step before the convergence criteria is reached, because the initial guess is better, like we discussed in Section 5.3.3.

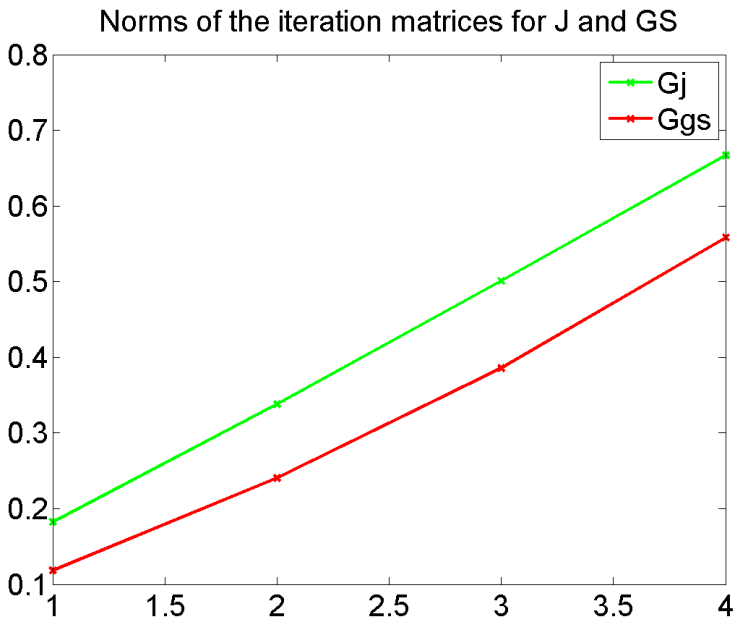


Figure 5.7: Norms of the iteration matrices for Jacobi's method (J) and Gauss-Seidel's method (GS). Relevant for the convergence rate of these two methods

	0.5	0.5^2	0.5^3	0.5^4
$\|\|\boldsymbol{G}_J\|\|$	0.18246	0.33793	0.50101	0.6669
$\sqrt{\|\|\boldsymbol{G}_J\|\|}$	0.42715	0.58131	0.70782	0.81664
$\|\|\boldsymbol{G}_{GS}\|\|$	0.11842	0.24043	0.38585	0.55826
$\sqrt{\|\|\boldsymbol{G}_{GS}\|\|}$	0.34412	0.49034	0.62117	0.74717

Table 5.6: Norms and square root of the norms of the iteration matrices for Jacobi's (J) and Gauss-Seidel's (GS) method when using Crank-Nicolson (CN)

Table 5.6 gives the norms of the iteration matrices for Jacobi's and Gauss-Seidel's method. Figure 5.7 shows a plot of these same norms. As this norm is what governs the convergence of Jacobi's and Gauss-Seidel's method, the methods will converge faster the smaller the norm is. We can use the same argument as we did in Section 5.3.4 to deduce how we expect the norms of the iteration matrices for Jacobi's method and Gauss-Seidel's method to behave. From Table 5.5 we see that the iteration numbers is approximately

doubled each time we refine the grid. This is the same behaviour as for Backward Euler. Therefore we expect the same relation between the iteration matrix at the current grid $\boldsymbol{G}_2$, and the previous grid $\boldsymbol{G}_1$, as we had for Backward Euler. This expected behaviour is

$$||\boldsymbol{G}_2|| \approx \sqrt{||\boldsymbol{G}_1||},$$

like we calculated in Section 5.3.4. We see from Table 5.6 that each norm is smaller than the square root of the previous norm, which is the expected behaviour.

As the number of iterations required for Richardson's method largely depends on the eigenvalues of the matrix $\boldsymbol{A}$ and these do not change when we use a different discretization scheme in time, the behaviour of this method is largely the same as it was for Backward Euler, as discussed in Section 5.3.4

5.3.6 Time Usage When Using Backward Euler in Time

Time is reported in seconds used by each method for solving the problem. As we saw when we discussed convergence rates, in Section 5.3.4, Richardson's method takes so much time it has not been run for higher than $h = 0.5^6$.

h	0.5	0.5^2	0.5^3	0.5^4	0.5^5	0.5^6	0.5^7
EE	0.0015263	8.1254e-05	0.00057049	0.012914	0.84082	31.642	958.26
R	0.38212	0.0029893	0.020352	0.81714	43.165	4415.0	-
J	0.0014262	0.00062822	0.0093981	0.1405	4.7133	177.55	4727.2
J_w	0.0017346	0.0037407	0.019421	0.27584	7.2296	301.67	8433.8
GS	0.0010353	0.0015011	0.0082289	0.10843	3.5848	125.45	3272.0
CG	0.00090748	0.001339	0.0038147	0.029359	0.38519	12.259	224.93
$BiCG$	0.20438	0.008358	0.0087369	0.030457	0.42058	13.787	270.32

Table 5.7: Time used by the different methods when using Backward Euler. EE is Explicit Euler and R, J and GS are Richardson's, Jacobi's and Gauss-Seidel's methods respectively. J_w is the weighted version of Jacobi's method, CG is the Conjugate-Gradient method and BiCG is the Biconjugate-Gradient method.

We see from Table 5.7 that Richardson's method uses much more time than the other iterative methods. This is because it requires so many more iterations to converge than the other two methods, as we saw in Section 5.3.4. What is worth noting is that to get the optimal rate of convergence we have to use the optimal α, $\alpha^* = \frac{2}{\lambda_{max}+\lambda_{min}}$, and to use this we must calculate the smallest and largest eigenvalue. As the matrix $\boldsymbol{A}$ gets large, this also becomes an increasingly time consuming operation. The time required to

calculate this is not included in the time usage for the Richardson's method, but this is a time consuming and necessary calculation that increases the time we have to use to solve the problem with this method.

To get an idea of how the time usage increases, we look at the relationship between the time usage for the current grid and the previous grid.

From the theory of the methods and the CPU model we discussed in Section 2.5, we can calculate what the time increase should be for the implicit methods. We know that for Explicit Euler the number of operations required is N^5 where N is the number of nodes in each direction. This comes from the fact that we get N^3 from the discretization in space and as Δt_E depends on $\frac{1}{h^2}$ we get another N^2 from this as $N = \frac{1}{h}$. For our grids we can easily use this to calculate the increase in number of operations. As we double the number of points in each direction whenever the grid is refined the increase in calculation time should be limited by $2^5 = 32$ according to the theory.

For the iterative methods the reasoning is a bit different. We get N^3 from the space discretization. We get an additional N from the time discretization as Δt for the iterative methods depends on h. In addition we have to take into account the number of iterations these methods require to converge so we also have to consider the increase in k as the grid is refined. To consider how the time usage is increased for the iterative methods we have to look at how kN^4 increases. As the grid increases at a set pace and doubles each time the grid is refined we know exactly how N^4 increases. So we know that the computation time should increase no more than $2^4k = 16k$ each time the grid is refined. As we can see in Table 5.3, discussed in Section 5.3.4, the methods have slightly different increases in required iterations. For both Gauss-Seidel's and Jacobi's method we see that the number of iterations is approximately doubled. This means that they should be limited by $16 \cdot 2 = 32$, like Explicit Euler. For Richardson's method we see from Table 5.3 that the situation is a bit different. For the coarsest grid this method does not converge, so studying the time used is not interesting. When it does converge we see that the increase in the number of iterations is larger than for the two other methods. It takes approximately 4 times as many iterations to converge for each time the grid is refined. So Richardson's method should be limited by $16 \cdot 4 = 64$.

	t_2/t_1	t_3/t_2	t_4/t_3	t_5/t_4	t_6/t_5	t_7/t_6
EE	0.0532	7.0211	22.6367	65.1092	37.6323	30.2844
R	0.0078	6.8083	40.1504	52.8245	102.2819	-
J	0.4405	14.96	14.95	33.5466	37.67	26.6246
GS	1.45	5.4819	13.1767	33.061	34.995	26.0821

Table 5.8: Time increase as the grid is refined for the various methods when using Backward Euler. EE is Explicit Euler and R, J and GS are Richardson's, Jacobi's and Gauss-Seidel's methods respectively. The subscript on the t denotes the exponential for the grid, ie t_6 is the time used to solve the problem for the grid $h = 0.5^6$.

As we see from Table 5.8 the practical tests of the problem seem to match what we expect based on theory. The exception is that the increase in time for Richardson's method is much higher than expected for t_6/t_5. We also see that Gauss-Seidel's method and Jacobi's method occasionally goes higher than expected. As it is difficult to accurately measure the computation time, this might be caused by MATLAB reporting that the methods have been using more time than they actually have been.

In Table 5.8 we see that from $h = 0.5^6$ and up the increase in time usage is getting smaller. The increase in time from $h = 0.5^6$ to $h = 0.5^7$ gets smaller than the previous increases. It has also gotten a lot smaller for both Jacobi's method and Gauss-Seidel's method than for Explicit Euler. If this trend continues then eventually the iterative methods will use less time to solve the problem than Explicit Euler. As long as we have the computational power, we are interested in solving the problem for as fine a grid as possible. The fact that the explicit method is so much better than the iterative methods for as coarse grids as we have been using does not necessarily mean that it will still be better than the iterative methods for much finer grids than the ones we have been able to solve for here. Ideally we should have had the computational power to solve for $h = 0.5^8$ as well to see if this trend continues, but we do not at the moment have access to the computational power to be able to do this.

5.3.7 Time Usage When Using Crank-Nicolson in Time

From Table 5.9 we see that the iterative methods use much less time for this second order scheme in time than they did for the first order scheme. The solving time has been approximately halved. The reason for this reduction is primarily that we with this method are able to use a larger Δt and still get accurate results. We take fewer time steps using this time discretization scheme.

h	0.5	0.5^2	0.5^3	0.5^4	0.5^5	0.5^6	0.5^7
EE	0.0015263	8.1254e-05	0.00057049	0.012914	0.84082	31.642	958.26
R	0.19236	0.0023174	0.0085753	0.34332	18.224	1980.1	-
J	0.0031437	0.00026643	0.0046323	0.068372	2.2663	92.949	2592.3
J_w	0.0011837	0.001951	0.0094734	0.12847	3.7019	152.05	4556.0
GS	0.00081981	0.00075652	0.0045772	0.052748	1.6179	64.626	1740.4
CG	0.00045844	0.00067441	0.0024171	0.015739	0.20854	7.7727	125.95
$BiCG$	0.0010828	0.0013779	0.0032908	0.015922	0.26673	11.705	193.01

Table 5.9: Time usage for the matrix versions of the methods when using Crank-Nicolson. EE is Explicit Euler and R, J and GS are Richardson's, Jacobi's and Gauss-Seidel's methods respectively. J_w is the weighted version of Jacobi's method, CG is the Conjugate-Gradient method and BiCG is the Biconjugate-Gradient method.

We see when using a second order scheme that Richardson's method uses much more time than the other iterative methods. This is again due to the fact that it also here needs many more iterations, as shown in Table 5.5.

The increase in time usage for this scheme, like for the first order scheme, is given in Table 5.10.

	t_2/t_1	t_3/t_2	t_4/t_3	t_5/t_4	t_6/t_5	t_7/t_6
EE	0.0532	7.0211	22.6367	65.1092	37.6323	30.2844
R	0.012	3.7	40.036	53.0817	108.6534	-
J	0.0848	17.3866	14.76	33.1466	41.0154	27.8895
GS	0.923	6.0503	11.524	30.3723	39.9444	26.93

Table 5.10: The increase in time as the grid is refined when using Crank-Nicolson. EE is Explicit Euler and R, J and GS are Richardson's, Jacobi's and Gauss-Seidel's methods respectively. The subscript on the t denotes the exponential for the grid, ie t_6 is the time used to solve the problem for the grid $h = 0.5^6$.

As the way we discretize the problem in time does not affect the expected increase in computation time, we can use the same argument as in Section 5.3.6. The theory gives us that the increase in computation time should be $16k$ for the implicit schemes, just like for Backward Euler. k is the increase in the number of iterations required for convergence. As for Backward Euler, both Jacobi's and Gauss-Seidel's methods need approximately twice the number of operations each time the grid is refined, as we see from Table 5.5. Because of this, both methods should not have an increase in time larger than 32. For Richardson's method we see that the increase is about 4 times more iterations so this method should not have a time increase of

more than 64. This is the same as the limits found in Section 5.3.6. Again we see that they fit rather well with the results. The only exception is that Richardson's method does have a much larger increase than expected when refining the grid from $h = 0.5^5$ to $h = 0.5^6$. This is the same behaviour as we saw when using Backward Euler. For this grid refinement we also see that both Jacobi's and Gauss-Seidel's method has a larger than expected increase in time usage, but the increase becomes much smaller again for the next grid refinement. The only result that does not fit the theory is again the increase in time from the grid with $h = 0.5^5$ to $h = 0.5^6$.

We see that while the total time used has decreased a lot for this time scheme, the increase in time used as the grid is refined has actually increased slightly. Here it would be ideal to be able to solve the problem also for $h = 0.5^8$ to see if the time increase keeps getting smaller and whether the time increase has then become smaller than for the explicit scheme. This is at the moment not possible as we do not have enough memory available.

Again we see that Richardson's method is not a very useful method for this problem. When the grid is refined the increase in time usage is much larger for this method than the other methods.

5.3.8 Errors When Using Backward Euler in Time

The errors are calculated by taking the norm of the numerical solution minus the analytical solution and dividing by the norm of the analytical solution, meaning

$$E = \frac{||\boldsymbol{v} - \boldsymbol{u}||}{||\boldsymbol{u}||},$$

where $\boldsymbol{v}$ denotes the numerical solution and $\boldsymbol{u}$ denotes the analytical solution. Richardsons method diverges for $h = 0.5$.

h	0.5	0.5^2	0.5^3	0.5^4	0.5^5	0.5^6	0.5^7
E_{EE}	13.923	0.99996	0.49392	0.14626	0.03757	0.00949	0.00238
E_{EE}/h	27.845	3.9998	3.9513	2.3401	1.2023	0.60733	0.30416
E_{EE}/h^2	55.691	15.999	31.611	37.442	38.473	38.869	38.933
E_R	-	5.5804	1.3856	0.49992	0.21392	0.09987	-
E_R/h	-	22.322	11.085	7.9987	6.8454	6.3918	-
E_R/h^2	-	89.286	88.677	127.98	219.05	409.08	-
E_J	48.547	5.5804	1.3855	0.49985	0.21385	0.09978	0.04977
E_J/h	97.095	22.322	11.084	7.9977	6.8431	6.3862	6.3711
E_J/h^2	194.19	89.286	88.674	127.96	218.98	408.72	815.5
E_{JW}	48.547	5.5805	1.3856	0.49988	0.21387	0.09981	0.04981
E_{JW}/h	97.095	22.322	11.084	7.9981	6.8439	6.3882	6.376
E_{JW}/h^2	194.19	89.287	88.675	127.97	219	408.84	816.13
E_{GS}	48.547	5.5804	1.3854	0.49962	0.21339	0.09887	0.04799
E_{GS}/h	97.095	22.322	11.084	7.9939	6.8283	6.3274	6.1424
E_{GS}/h^2	194.19	89.286	88.668	127.9	218.51	404.95	786.22
E_{CG}	98.841	30.062	7.9874	2.5209	0.96319	0.4188	0.19532
E_{CG}/h	197.68	120.25	63.899	40.334	30.822	26.803	25.001
E_{CG}/h^2	395.36	480.99	511.19	645.35	986.3	1715.4	3200.1
E_{BiCG}	48.547	5.5804	1.3854	0.49963	0.21332	0.09867	0.04746
E_{BiCG}/h	97.095	22.322	11.084	7.994	6.8263	6.3147	6.0748
E_{BiCG}/h^2	194.19	89.286	88.668	127.9	218.44	404.14	777.57

Table 5.11: Errors for the various methods when using Backward Euler. EE is Explicit Euler and R, J and GS are Richardson's, Jacobi's and Gauss-Seidel's methods respectively. JW is the weighted version of Jacobi's method, CG is the Conjugate-Gradient method and BiCG is the Biconjugate-Gradient method.

We see from Table 5.11 that all the implicit methods give approximately the same errors when compared to the analytical solution. So at least for this problem we can not use accuracy to determine if one method is better than the others. We could ensure that all the classical iterative methods gave the exact same error by using a small enough tolerance. The trade off for this is that they will require much more time to solve the problem. As all the methods give small enough errors to be useful, it was more useful for our purpose to use as little time as possible when solving the problem with a suitable accuracy instead of solving the problem as accurately as possible.

For some grids Richardson's method has not been allowed to converge, as this requires to many iterations. We see that the error for this method is almost the same as for the other iterative methods when it has converged, so it is natural to assume that it will also give the same error for the grids where we have not given it the time to converge. We have not attempted to calculate the error for Richardson's method for $h = 0.5^7$ as the time usage required for calculating the eigenvalues and iterating to convergence would

be too large.

It is important to note that as we use a normalized error the error must be less than 1, because as long as the normed error is larger than 1 the solvers give so inaccurate solutions that they are no longer useful. We see that the implicit methods do not get an error smaller than 1 before we use a grid with $h = 0.5^4$. However the most important result to take away from these error measurements is that the error is steadily decreasing for all methods. This means that for all coarser grids than this there is really no point in comparing Explicit Euler and the implicit methods, as the implicit methods are too inaccurate. This stems from the fact that we use a much larger time step for the implicit methods, where Δt depends on h, compared to the explicit method, where Δt depends on h^2.

Like mentioned at the start of Section 5.3, we can solve the implicit methods for a smaller Δt. This will result in all the implicit methods giving more accurate solutions. If we use a small enough Δt the implicit methods will become more accurate than Explicit Euler. The reason for not using such a small Δt is that this will cause a large increase in the time the implicit methods uses to solve the problem. As these methods are already much slower than Explicit Euler we have to balance the accuracy against the efficiency of the methods. It is not necessarily our goal to get the implicit methods as accurate as possible. We want the methods to solve the problems for as few time steps as possible, while still being accurate enough to give a valuable solution.

As the accuracy increases when Δt decreases, this means that the finer the grid the more accurate the solution, which is what we should expect. We see, from Table 5.11, that the error is steadily becoming smaller. Therefore it is natural to assume that finer grids than the ones we have solved for will all give accurate enough solutions. While we see that the Explicit Euler scheme is more accurate, we also know that the error for the implicit methods will probably be accurate enough for grids from $h = 0.5^6$ and up.

We see that the methods obviously converge towards the analytical solution with more than first order convergence. While for $\frac{E}{h^2}$ the values increases so we do not have second order convergence. This is because we solve the problem by discretizing using backward Euler in time which is a first order scheme. It is very likely that we will get second order convergence,or at least much closer to second order convergence, if we use a second order discretization in time. In the next section, Section 5.3.9, we look at the errors for the Crank-Nicolson scheme to see if this is the case.

3-dimensional plots of the error when subtracting the analytical solution from the numerical solution obtained by solving the problem for various

methods are given in Figure 5.8.

The plots in Figure 5.8 show the error in the plane $z = 0$ for the finest grid $h = 0.5^7$, except for Richardson's method which is for $h = 0.5^6$. As we can see the maximum and minimum values for this error is much smaller than the normed errors we have discussed so far.

There is a very significant difference between the error for Explicit Euler and the iterative methods in the plots in Figure 5.8. While the error for Explicit Euler is of order 10^{-5} the error for the iterative methods is of order 10^{-4}. This is a significant difference. This echoes the fact that the normed errors were so much smaller for the explicit method compared to the implicit methods.

As we know from the theory of differential equation solvers it is common that Backward Euler gives a solution a bit larger than the analytical solution and Explicit Euler gives a solution that is a bit smaller. This is the reason the error plot for explicit Euler has the opposite sign of the others.

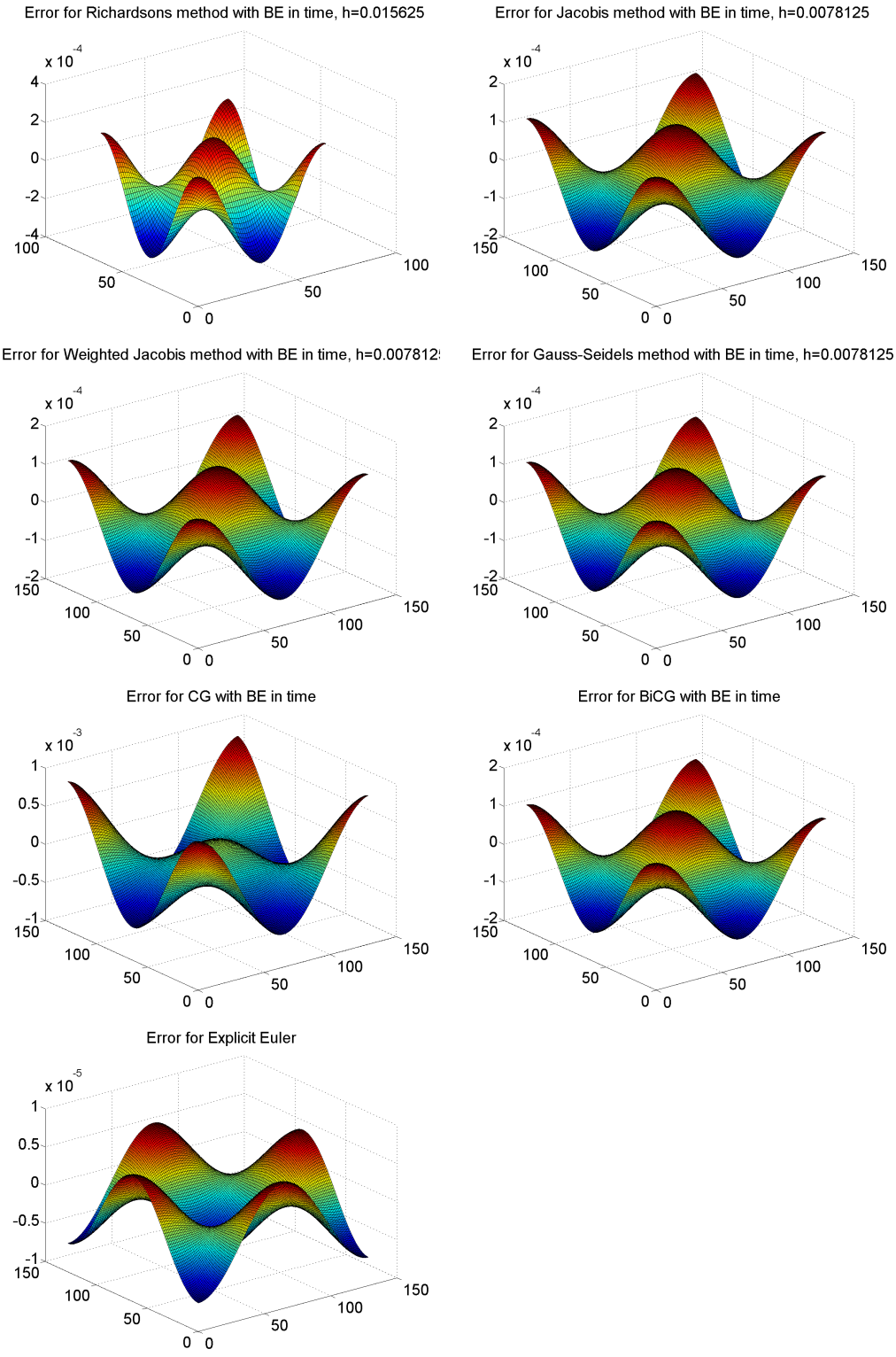


Figure 5.8: Errors for the various methods in the plane $z = 0$ with Backward Euler

5.3.9 Errors When Using Crank-Nicolson in Time

h	0.5	0.5^2	0.5^3	0.5^4	0.5^5	0.5^6	0.5^7
E_{EE}	13.923	0.99996	0.49392	0.14626	0.03757	0.00949	0.00238
E_{EE}/h	27.845	3.9998	3.9513	2.3401	1.2023	0.60733	0.30416
E_{EE}/h^2	55.691	15.999	31.611	37.442	38.473	38.869	38.933
E_R	-	0.99757	0.15328	0.03424	0.00798	0.00233	-
E_R/h	-	3.9903	1.2262	0.5176	0.255	0.1488	-
E_R/h^2	-	15.961	9.8098	8.7642	8.1586	9.5243	-
E_J	28.367	0.99755	0.15327	0.03421	0.00794	0.00230	0.00140
E_J/h	56.734	3.9902	1.2261	0.54741	0.25406	0.14728	0.17887
E_J/h^2	113.47	15.961	9.809	8.7586	8.1301	9.4262	22.896
E_{JW}	28.367	0.99758	0.15326	0.03423	0.00796	0.00231	0.00141
E_{JW}/h	56.734	3.9903	1.2261	0.5476	0.25475	0.14773	0.18013
E_{JW}/h^2	113.47	15.961	9.8086	8.7616	8.1519	9.4546	23.056
E_{GS}	28.367	0.99755	0.15322	0.03411	0.00775	0.00193	0.00068
E_{GS}/h	56.734	3.9902	1.2258	0.5457	0.24799	0.12319	0.08693
E_{GS}/h^2	113.47	15.961	9.8063	8.7312	7.9356	7.8842	11.127
E_{CG}	83.742	23.641	6.1131	1.9487	0.76054	0.33703	0.15914
E_{CG}/h	167.48	94.564	48.905	31.179	24.337	21.57	20.37
E_{CG}/h^2	334.97	378.26	391.24	498.86	778.8	1380.5	2607.4
E_{BiCG}	28.367	0.99755	0.15322	0.03411	0.00772	0.00184	0.00046
E_{BiCG}/h	56.734	3.9902	1.2258	0.54582	0.24715	0.1179	0.05884
E_{BiCG}/h^2	113.47	15.961	9.8061	8.7331	7.9087	7.5457	7.5315

Table 5.12: Normed errors for the methods when using Crank-Nicolson. EE is Explicit Euler and R, J and GS are Richardson's, Jacobi's and Gauss-Seidel's methods respectively. JW is the weighted version of Jacobi's method, CG is the Conjugate-Gradient method and BiCG is the Biconjugate-Gradient method.

When we use a second order discretization scheme in time we see from Table 5.12 that the errors are much smaller than when we used a first order scheme, as in Table 5.11. Now the error for the implicit methods is smaller than the error given by Explicit Euler. Due to discretization, there is always some error when we use numerical methods. Therefore the iterative methods do not necessarily have to give the exact solution. The method simply has to be accurate enough.

The normed error we have used to get the results in Table 5.12 is the same as the one discussed in Section 5.3.8. As before, this error has to be below 1 for us to even consider the accuracy of the numerical solution and for the solution to be accurate enough for our purposes the normed error should be much smaller than 1. While this error does not get smaller than 1 until $h = 0.5^4$ when using Backward Euler this happens already for $h = 0.5^2$ for Crank-Nicolson. The error has become so small that we can discuss having

obtained accurate enough results when we reach $h = 0.5^4$.

We also see that the errors are steadily decreasing for the second order scheme as well, like we saw for the first order scheme, but now it decreases much faster than it did for the first order scheme. The error for the implicit scheme also decreases faster than for the explicit scheme, so the iterative methods now gives much more accurate results than Explicit Euler.

While we in Table 5.11 saw that Backward Euler discretization gives us more than first order convergence, as $\frac{E}{h}$ steadily decrease, we were still far away from second order convergence, as $\frac{E}{h^2}$ increased rapidly. From Table 5.12 we see that the convergence rate of the methods have been vastly improved by using a second order scheme. Now we see that even $\frac{E}{h^2}$ is decreasing for the iterative methods until it stabilizes. Unfortunately it increases again for $h = 0.5^7$. This increase could possibly be nullified by a stricter tolerance criteria. The disadvantage of this is that it would increase the time used, especially if we have to use a much stricter tolerance for convergence. From Table 5.12 we see that using a second order scheme gives us second order convergence towards the analytical solution, or at least close to second order convergence. This is expected behaviour as it is natural to assume that a first order scheme will give first order convergence and a second order scheme gives second order convergence. Based on the fact that the two discretization schemes we have used to discretize in time are of different order it is also expected that the second order scheme should give smaller errors and better convergence than the first order scheme. We see that this expectation fits the results we have obtained.

Here again we have the same issues with Richardson's method, as in Section 5.3.8. It diverges for the coarsest grid and is too time consuming to calculate for the finest grid. However we also see that the errors when it does converge are almost identical to the two other methods. Therefore it is natural to assume also here that Richardson's method is just as accurate as the two other methods. It just takes a lot longer to get there. Here like for Backward Euler we could get all three classical iterative methods to give the exact same solution if we use a strict enough tolerance criteria.

Plots of the error when using Crank-Nicolson in time are given in Figure 5.9. The plot for Explicit Euler has also been included in this figure for easy comparisons.

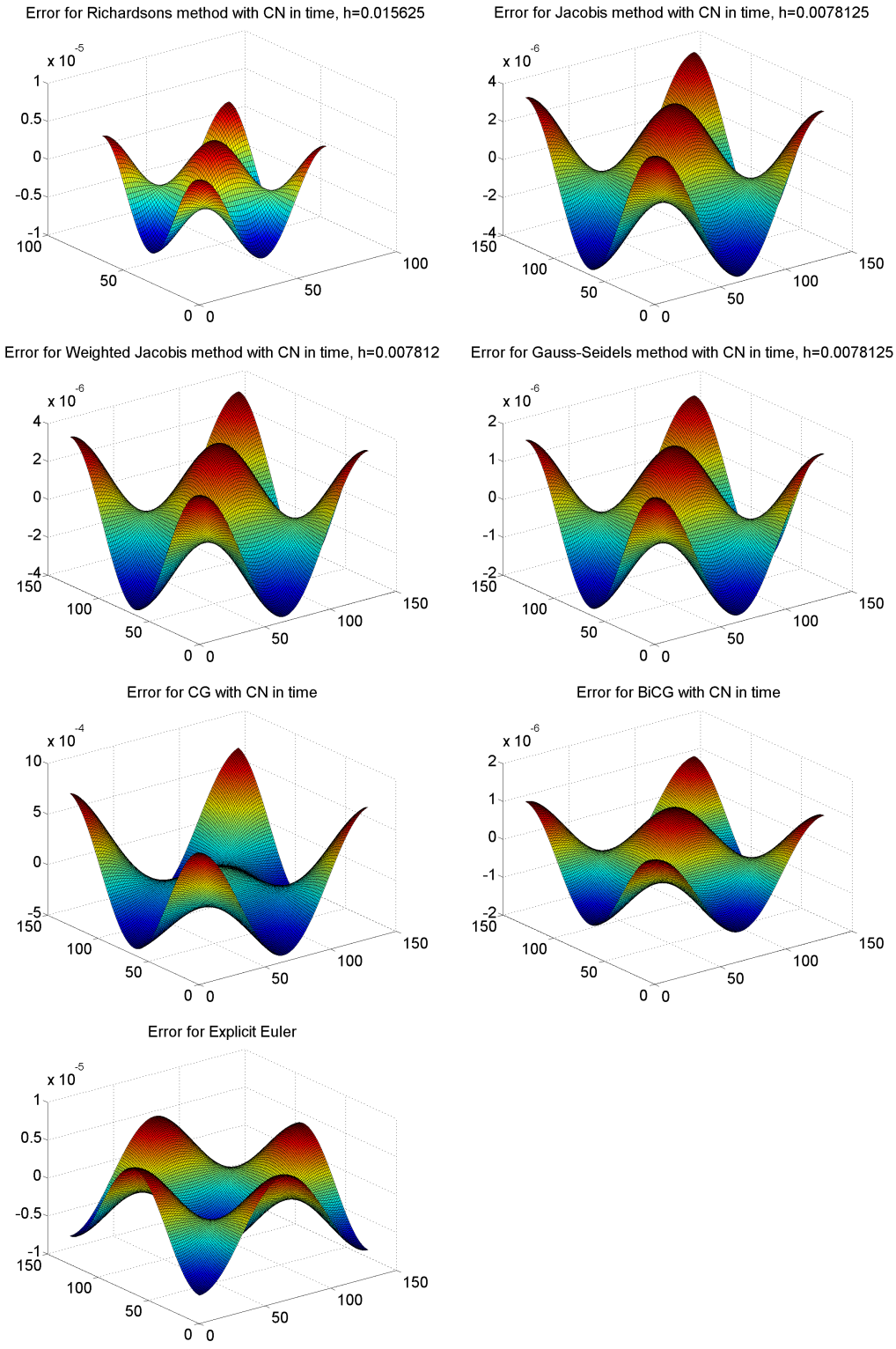


Figure 5.9: Errors for the various methods in the plane $z = 0$ with Crank-Nicolson

When using a second order scheme to discretize in time we also see a significant change in the error in the plane $z = 0$. Like we saw when discussing

the normed error, the error for the iterative methods has become significantly smaller and is now smaller than for the explicit method. The error for the iterative methods is of order 10^{-6} compared to order 10^{-4} as it was for Backward Euler. The iterative methods now give an error of an order smaller than Explicit Euler instead of an order larger like it was when using the first order scheme. Explicit Euler is still at order 10^{-5} as this scheme is not affected by which implicit time discretization we use.

5.3.10 Using Different Initial Guess Vectors

For all the results we have discussed so far we tried to make the best possible choice of initial guess vector every time we start the iteration for the iterative methods. We have used the result vector computed for the previous time step as the initial guess vector for the current time step. Using different initial guess vectors have varying results. Going with the standard choice of using the zero vector gives some increase in the number of iterations required for each time step and therefore some increase in time used. This increase gets larger the finer the grid is, but it is not so large that it greatly effects our results. Using vectors with randomly generated integers does however give much worse result. When using vectors like these the number of iterations is more than doubled and this causes a large increase in the time used. Especially for the finer grids. We therefore see that for our test problem it is important to make a good choice of initial guess vector or else the time usage might become discouragingly large. Using the result from the previous time step or using the zero vector does however give good results. The results are better when using the previous result vector than the zero vector.

5.3.11 Variants of Jacobi's Method

There are variants of Jacobi's method which will, for certain problems, give improved results compared to the regular Jacobi's method. One of these methods is the Weighted Jacobi method. As discussed in Section 4.4.2, this method does not accelerate the Jacobi iteration. It works much better as a smoother. According to Saad (2003), the Weighted Jacobi's method will give the fastest results when $\omega = 1$, which is the regular Jacobi's method, while the best results as a smoother will be obtained with $\omega = \frac{2}{3}$. We have included the results from Weighted Jacobi with $\omega = \frac{2}{3}$ throughout this chapter, to see if our solvers could benefits from using a smoother. As we can see from Table 5.11 and Table 5.12, this Weighted Jacobi method gives less accurate results than the unweighted version. Based on this it seems that a smoother is not necessary for our test problem. This could change if we used an initial condition which is more difficult to approximate nu-

merically than the one we have used so far. We discuss this in Section 5.3.12.

In the article Yang and Mittal (2014) the authors present an improved version of the Weighted Jacobi method. This method and the improvements it gives with respect to iteration numbers are presented in relation to elliptic systems of equations. For our problem we can get quite a lot of improvement to the convergence rate by adjusting Δt to find an optimal relation between speed and accuracy. This is an option which is not available for elliptic equations and these types of problems generally have much slower convergence rates than time dependent problems. As such our test problem already lends itself much better to being solved by iterative methods than elliptic problems before we have even started optimizing the iterative method. As we have seen from our results using the Weighted Jacobi method, our time dependent test problem actually gives better results without weights than with. The method presented in Yang and Mittal (2014) requires us to calculate several parameters to achieve optimal convergence rates as it is unlikely that the values presented in the article will be the optimal choices for our problem since they are calculated for a different type of problem. Because we get such good results using the regular version of Jacobi's method for our problem and because the weighted version gave worse results we have not calculated these parameters and implemented this version of the method for our test problem as it is unlikely that it will give significant improvements.

Based on the results we have obtained in our tests we see that for our test problem we should use the regular version of Jacobi's method and not the weighted version.

5.3.12 Other Initial Conditions

The test problem we have been solving has the initial condition

$$\boldsymbol{u}_0 = \cos(2\pi x)\cos(2\pi y)\cos(2\pi z).$$

This initial condition results in only low frequent errors and it is therefore easy for our methods to approximate the solution.

We also tried solving problems where we use different initial condition vectors, but otherwise the problem remains the same. For instance we used

$$\boldsymbol{u}_0 = \begin{cases} 0, & 0 \leq x < \frac{1}{2} \\ 1, & \frac{1}{2} \leq x \leq 1 \end{cases}$$

which is the initial condition which gives the most high frequent errors, to see how our methods behave with these types of errors.

The results from these tests where much as the results we have reported

previously in this chapter. The more high frequent errors the initial condition gave, the more iterations we needed to reach our convergence criteria and therefore it takes more time to solve the problem. But the trends remained the same. The Conjugate-Gradient method and Biconjugate-Gradent method are much better than the other methods. Explicit Euler still gives better results than the classical iterative methods and Gauss-Seidel is still the best of these methods. Surprisingly enough Weighted Jacobi's method still requires more iterations than Jacobi's method even when we have primarily high frequent errors which are the types of errors that Weighted Jacobi's method is good at removing.

5.3.13 Results for the Conjugate-Gradient Method

We have noted the results when solving the problem with the Conjugate-Gradient method, but so far we have not discussed them. We see from the results in Section 5.3.6 and Section 5.3.7 that this method is much more efficient than the classical iterative methods. This is due to the fact that it only needs a fraction of the iterations to converge. As such it appears to be a very efficient and useful method. But there are some rather large drawbacks to this method.

One is that it is very difficult to parallelize this method in an efficient way and as our goal here is to ultimately use the results to solve large problems where we must take advantage of parallelization, this is a disadvantage. We discuss the problems that arise when using the Conjugate-Gradient method in parallel in Section 6.5.

Another problem is the question of accuracy. Because the Conjugate-Gradient method only works on symmetric positive definite matrices we had to adjust the discretization of our test problem to use first order boundary conditions as the second order conditions we used for the other methods broke symmetry. This resulted in the Conjugate-Gradient method giving a less accurate numerical solution compared to the other iterative methods. It does still give a solution that is accurate enough, but when we start to compare it to the solution given by Explicit Euler the gap is getting rather large. This is especially true when we use Crank-Nicolson to discretize in time. If we look at Table 5.12 we see that for $h = 0.5^7$, the error for the Conjugate-Gradient method is 0.15914 while the error for the classical iterative methods is as small as 0.00140. We also see that the error for the classical iterative methods decreases a lot faster than the error for the Conjugate-Gradient method. The reason is that this method has a much slower rate of convergence towards the analytical solution. Because we have to use first order boundary conditions the rate of convergence has been considerably reduced. This means that for the Conjugate-Gradient method the rate of convergence is first order, while for the other methods we are

closer to second order than to first order convergence when using Backward Euler and we have second order convergence when using Crank-Nicolson. As the grid gets more and more refined this will result in a larger and larger difference between this method and the classical iterative methods.

When we look at the plots of the errors in the plane $z = 0$ we also see that it is around the edge of the plane, and especially in the corners that this method struggles. This corresponds well to the fact that using first order boundary conditions reduces the accuracy when solving this problem.

We have also looked at results for the Biconjugate-Gradient method. This is a version of the Conjugate-Gradient method. This method keeps the advantageous properties of the Conjugate-Gradient method, like the fast convergence, but it does not require the matrix to be symmetrical. This means that we can apply the Biconjugate-Gradient method to the same matrix that the classical iterative methods use. As such this method gives us the much faster convergence, and thus lower time usage, of the Conjugate-Gradient method combined with the higher accuracy and faster convergence to the analytical solution we get from using second order boundary conditions. As we see from the various result tables presented in the previous sections we get very good results, as expected. From Tables 5.7 and 5.9 we see that this method, like the Conjugate-Gradient method, uses only a fraction of the time compared to what the classical methods use. This is primarily caused by the fact that this method use fewer iterations, as we can see from Tables 5.3 and 5.5. When we look at the errors of this method we see from Tables 5.11 and 5.12 that it actually gives more accurate results than the classical iterative methods. Thereby the Biconjugate-Gradient methods gives better results than the Conjugate-Gradient with fewer drawbacks. But this method is at least as badly suited for parallellization as the Conjugate-Gradient method.

5.4 Summary

5.4.1 The Iterative Methods

For this problem it appears that Gauss-Seidel's method is the most effective classical iterative method for solving the partial differential equation. This method uses the shortest time, has the smallest increase in the number of operations and needs the fewest iterations to converge. So for our problem and implementation it is the most suited solver. Because of the structure of this method it might however be difficult to effectively program in parallel. Jacobi's method is not so far behind Gauss-Seidel's method when it comes to time usage and Jacobi's method will probably be better suited for parallelization than Gauss-Seidel's method. So in conclusion both Gauss-Seidel's method and Jacobi's method seems to be good candidates for further in-

vestigation. They give fast and accurate results for this particular problem. Gauss-Seidel is faster, but will run into problems when parallelization is taken into account. We discuss parallelization of these two methods in Section 6.3

When comparing the iterative methods we find that Richardson's method is not very useful because the number of iterations it requires, and as a result the time used, rapidly increases as the mesh is refined. This method gives just as accurate results as the other two, but we have to wait considerably longer for it.

5.4.2 The Time Discretization

The test results show some significant differences between using a first order scheme and a second order scheme to discretize in time. When using Backward Euler the results from the implicit methods are far behind the explicit method, both in accuracy and time consumption. The error for the iterative methods does not become small enough to discuss accuracy until the grid is at $h = 0.5^4$ and for $h = 0.5^7$ the error is still large compared to the error from Explicit Euler. Also the time usage is so much larger for the implicit methods that the grid will have to be refined a lot more before we can even start to compare the methods when it comes to efficiency.

However when using Crank-Nicolson the results become quite different. Now we reach the minimum requirement of accuracy already at $h = 0.5^2$ and as the grid is refined the error for the implicit methods become significantly smaller than Explicit Euler. So the iterative methods are much more accurate than the explicit method. We also see that because we can use a larger time step compared to the first order scheme, the time usage has also significantly decreased for the iterative methods. While Explicit Euler is still faster it is possible to see that the iterative methods might eventually be able to compete with this explicit method when the grid becomes fine enough. Therefore it would seem that this second order time discretization scheme is the only one worth considering for further investigation.

5.4.3 Methods Considered for Parallelization

From the results we have presented in this chapter we can rule out Richardson's method. This method requires so much time to converge that it will probably never be able to compete with Explicit Euler. It is therefore no point in looking at this method when discussing parallelization. We also see that using Crank-Nicolson to discretize in time gives much better accuracy and time usage than Backward Euler so going forward we only need

to look at this second order time discretization. We also need to look at theory for parallelization before we can determine which one of Jacobi's and Gauss-Seidel's method is the most suitable classical iterative method, but it is very probable that Jacobi's method will work better than Gauss-Seidel's method when using parallel programming. We discuss the communication needs of a parallel implementation of both these methods in Section 6.3.

So the serial results gives that we need to look primarily at Jacobi's method and see if it will be able to compete with Explicit Euler and the Conjugate-Gradient method when using a second order time discretization scheme and parallel implementation.

For our parallel implementation we will consider the Conjugate-Gradient method and not the Biconjugate-Gradient method. This is because we will be adapting a pre-existing code for our parallel tests. In this code the Conjugate-Gradient method is already implemented. As we saw from the results in this chapter there is not much difference between these two methods, so this choice should not affect the main findings in the parallel tests. The Biconjugate-Gradient method is more accurate, but slightly slower.

Chapter 6

Parallelization of the Iterative Methods

6.1 Introduction

As we generally are interested in solving very large systems of linear equations it is useful to be able to utilize parallelization. When using iterative methods to solve linear problems in parallel we have to look at several issues when considering which method is the best. In addition to the computation time for each method, we have to consider how much communication between processes the various methods need. Because we want to compare the results from using these solvers in parallel with the results we got from our serial tests, in Chapter 5, we will solve the same test problem in parallel as we did in serial.

6.2 Which methods should we consider?

Based on the results from the serial tests there are a couple of methods that gave such poor results we do not need to consider them when discussing the parallel implementation, hence the discussion in Section 5.4.3. Because Richardson's method requires so much computation time compared to the other classical methods it is very unlikely that it will be able to catch up to the Conjugate-Gradient method (CG) and Explicit Euler when implemented in parallel. We will therefore not discuss this method any further. We saw, in Chapter 5, that using Crank-Nicolson instead of Backward Euler to discretize in time gave better results. Therefore we will also only look at Crank-Nicolson when we test the parallel implementation.

6.3 Gauss-Seidel or Jacobi?

We are interested in comparing the classical iterative methods with Explicit Euler and CG. We have chosen to implement only one of the classical it-

erative methods, as the results for Jacobi's and Gauss-Seidel's method are similar and as it is a time consuming task to implement them in parallel. As Jacobi's method is simpler than Gauss-Seidel it unsurprisingly did not give as good results. The results from Jacobi's method are however close enough to Gauss-Seidel's results that we cannot discount it outright. Because of this we have to consider how much communication each method requires before choosing which one to implement as communication can possibly be the most time consuming part of running parallelized programs.

6.3.1 Communication for Jacobi's method

Due to the simplicity of Jacobi's method, it is relatively simple to adapt it from a serial to a parallel implementation. Section 4.4 described that this method only depends on values calculated in the previous iteration. So if we have N equations and access to N processors we could give each process one equation each and solve all the equations simultaneously.

There are two things that requires communication between processes for this method. One is to check if the method has converged. This will require all the processes to communicate with each other. This is an unavoidable step for any iterative method when implemented in parallel. As this will usually involve comparing which process has the largest difference for some convergence criteria, this will usually be implemented as an `AllReduce` statement which will be almost identical regardless of which iterative method is used.

The other part of this method that requires communication is before we solve for each iterative step. How much communication is needed here will depend on how the method is implemented. If each process controls their part of the matrix $\boldsymbol{A}$ and the right hand side vector $\boldsymbol{b}$, we only need to send parts of $\boldsymbol{u}^{(k-1)}$ between processes. Otherwise we might have to send parts of $\boldsymbol{A}$ and $\boldsymbol{b}$ around as well. For $\boldsymbol{u}^{(k-1)}$, and $\boldsymbol{b}$ if we need to share this as well, the simplest and least efficient way of sharing the information is if each process sends their part of the vectors to all the other processes so all processes have a complete copy of each vector. The other possibility is to identify which processes each process needs information from and code it so that each process only communicates with the processes they need information from.

6.3.2 Communication for Gauss-Seidel's method

A major problem with parallelization of Gauss-Seidel's method is related to the fact that we need to know the first $i-1$ values of $\boldsymbol{u}^{(k+1)}$ to calculate $\boldsymbol{u}_i^{(k+1)}$. This causes this method to be very serialized. It is possible to use this method in parallel by using a version called the Blockwise Gauss-Seidel method. This uses Gauss-Seidel's method on the internal nodes each process

controls, but starts with only using old values for the first equation solved. This will of course cause this method to loose some of the advantage it has over Jacobi's method. As the number of processes increases the number of equations solved using only values from the previous iteration increases. Depending on how we divide the equations between the processes we would probably also be missing necessary values at the current iterations step because each process most likely do not have a continuous set of rows in the $\boldsymbol{A}$-matrix. This is a problem because Gauss-Seidel's method uses the first $i-1$ values of $\boldsymbol{u}^{(k+1)}$ to calculate $\boldsymbol{u}_i^{(k+1)}$ and they might not necessarily all be available. There are two ways to solve this. The first is that each process needs to communicate their calculated value of $\boldsymbol{u}_i^{(k+1)}$ for each i to the processes that need this value. This will increase the amount of communication needed and thus increase the time used. Another option is to use the value calculated at the previous time step when we know that the value for the current time step is unavailable. The disadvantage of the last option is that the more often we have to do this the closer we get to Jacobi's method and we will thereby loose the advantage Gauss-Seidel's method has over Jacobi's method when it comes to accuracy and rate of convergence.

6.3.3 Chosen Method

From the serial implementation we know that Jacobi is just as accurate as Gauss-Seidel and it is not very far behind when it comes to time usage for our particular problem. We choose to implement Jacobi's method in parallel and not Gauss-Seidel. This is because we know that Gauss-Seidel's method needs at least as much communication as Jacobi's method, if not more. We also saw that for Gauss-Seidel's method we have to choose between loosing convergence rate compared to the serial case or spending much more time on communication compared to Jacobi's method. Jacobi's method is also a simpler method to implement using parallel programming than Gauss-Seidel's method.

6.4 Implementing Jacobi's method

As discussed in Section 4.4 Jacobi's method is a relatively simple method when we look at the computation needs. Therefore the primary challenge when implementing this method in parallel is sharing the necessary information between processes in an efficient way. This will generally be an issue at two points in the calculations. First we need to ensure that all processes have the parts of the $\boldsymbol{A}$ matrix and the $\boldsymbol{u}$ and $\boldsymbol{b}$ vectors so that the calculation can be performed. Then we need to find a communication effective way of checking if convergence has been reached.

6.4.1 Distributing the information

We will first discuss the issue of distributing the necessary information to each process so they can compute their part of $\boldsymbol{u}$. The way our code is implemented ensures that each process already has the parts of $\boldsymbol{A}$ and $\boldsymbol{b}$ that it needs for carrying out the part of the computation the process is responsible for. The only data we need to distribute is the solution from the previous iteration step, $\boldsymbol{u}_{prev}$. There are several ways to go about this.

The simplest, but most communication heavy, way would be if each process sent their part of $\boldsymbol{u}_{prev}$ to all the other processes. Then each process would have the complete vector and we would be guaranteed that each process has the necessary data. This method would not be very effective as it would require a lot of One-to-All or All-to-All communication and communication between processes is typically the most time consuming part of parallel programming. This will also increase the storage needed for each process as they all have to store large parts of $\boldsymbol{u}$ that they will never use. Pseudocode for this alternative is shown in Algorithm 1.

Algorithm 1 Pseudocode for sharing the entire $\boldsymbol{u}_{prev}$ with all processes when solving Jacobi's method.

```
for each iteration k do
    for each other process p do
        Send local u_prev to p
        Receive part of u_prev from p
        Store received part of u_prev in correct order
    end for
    Solve Jacobi's method
end for
```

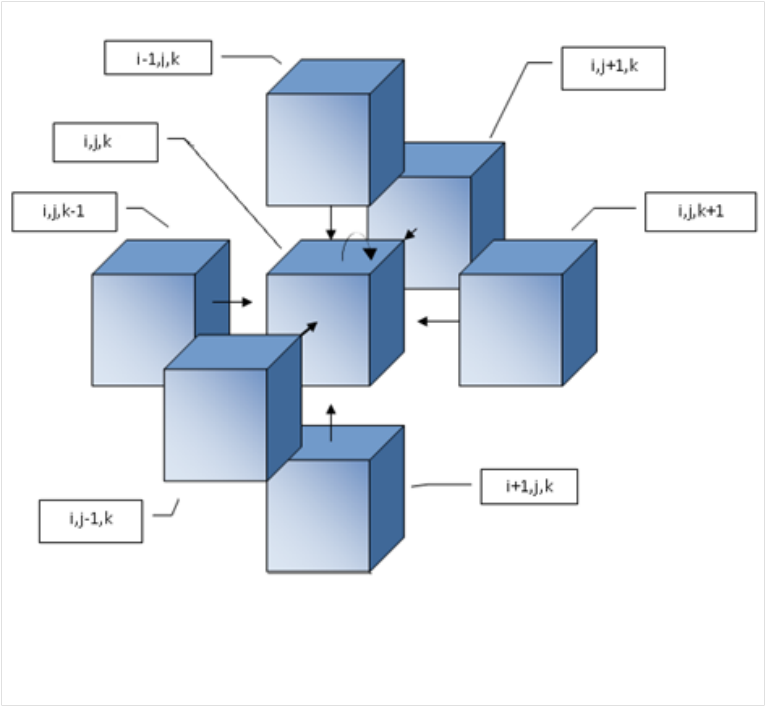
Another approach would be to just send each process the parts of $\boldsymbol{u}_{prev}$ that each process needs. This is difficult to do without hard coding the method to solve a specific kind of problem. As we are interested in solving a three-dimensional problem we have written our version of Jacobi's method so it solves 7-point stencils from finite difference schemes as presented in Section 3.2.2. Based on this knowledge we know that each process only needs information from a maximum of six other processes as long as the nodes are uniformly distributed. The reason we know this is based on how the values used in the stencil is structured. To solve Jacobi's method for the point $\boldsymbol{u}(i,j,k)$ we need the values of $\boldsymbol{u}_{prev}(i-1,j,k)$, $\boldsymbol{u}_{prev}(i+1,j,k)$, $\boldsymbol{u}_{prev}(i,j-1,k)$, $\boldsymbol{u}_{prev}(i,j+1,k)$, $\boldsymbol{u}_{prev}(i,j,k-1)$ and $\boldsymbol{u}_{prev}(i,j,k+1)$. For all the points that are internal to each process, that process has all the information needed to compute $\boldsymbol{u}(i,j,k)$. Along the edges of the area the process controls there will be at least one value missing. So each process

needs the values located in the surface bordering its own area from each of the neighbouring processes.

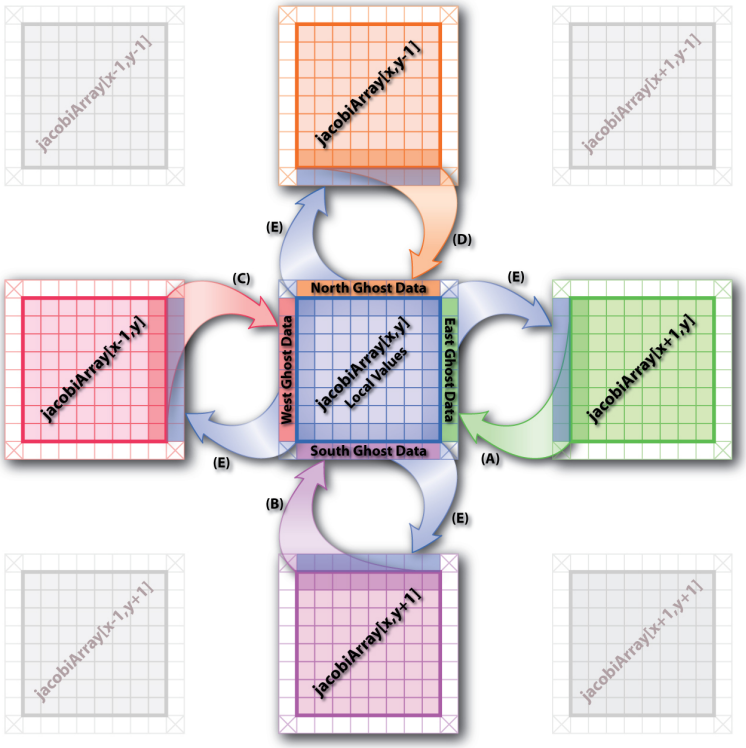
When solving a 3D-problem we know that a process will have a maximum of six bordering surfaces it needs values from and therefore a maximum of six processes it needs to exchange parts of $\boldsymbol{u}_{prev}$ with, as illustrated in Figure 6.1.

Using the method of only sending the needed parts of $\boldsymbol{u}_{prev}$ to the neighbouring processes compared to simply sharing the entire $\boldsymbol{u}_{prev}$ with all processes will require quite a lot of temporary storage. This is because the values of $\boldsymbol{u}_{prev}$ that has to be exchanged between neighbouring processes are not stored sequentially. Because of this we need several arrays for each neighbour the process has to exchange information with. We need one array where we calculate the indices for the values we need to send over. This array is used to get these values from $\boldsymbol{u}_{prev}$ and store them in a temporary array containing only the values of $\boldsymbol{u}_{prev}$ which has to be sent over to that particular neighbour. Then we receive an array with values from this neighbour as well so we have to store this information in an array. We also need an array containing the indices that these received values should be stored at in $\boldsymbol{u}_{prev}$ as well. So in total this method requires four arrays for each neighbouring process.

Two arrays contain indices. The elements in these arrays does not change as we iterate so we only need to define these once. The other two arrays are buffers for information sent to and received from the other process. These buffers have to be updated for each iteration. Using this method of less communication will require a lot more accessing temporary storage. But as communication is probably the most time consuming you can do when using parallel programming this is still better than the One-to-All communication that is the alternative. Once all needed information has been exchanged each process can solve Jacobi's method for their equations just like for a serial implementation. When this is done we run into the next communication problem which is checking if the method has converged. Pseudocode for only communicating with the neighbouring processes is found in Algorithm 2.



(a)



(b)

Figure 6.1: Illustrations of the communication needed between processes. 6.1a illustrates the 3D communication. This figure is taken from www.prace-ri.eu (2015). 6.1b illustrates 2D communication for Jacobi's method. This figure is taken from charm.cs.illinois.edu (2015).

Algorithm 2 Pseudocode for only sharing needed data with neighbouring processes when solving Jacobi's method.

```
Calculate process id of all neighbouring processes
for each neighbour np do
    Calculate indices of values of u_prev to send to np
    Calculate indices for storing values received from np
end for
for each iteration k do
    for each neighbour np do
        Store correct values of u_prev to send to np in a send buffer
        Send values to np and receive values from np
        Store received values at correct indices in u_prev
    end for
    Solve Jacobi's method
end for
```

6.4.2 Checking for convergence

There are two reasonable measures of convergence. Either we can look at the difference between the current and previous iteration and assume the method has converged if this difference is sufficiently small:

$$\max_i |\boldsymbol{u}(i) - \boldsymbol{u}_{prev}(i)| < \varepsilon,$$

or we can use the residual like we did for the serial implementation. As we know for the analytical solution we have that $\boldsymbol{b} - \boldsymbol{A}\boldsymbol{u}_{an} = 0$. We can thereby assume that the method has converged when this residual is smaller than the tolerance ε:

$$\max_i |\boldsymbol{b}(i) - \sum_j \boldsymbol{A}_{i,j}\boldsymbol{u}(j)| < \varepsilon.$$

Regardless of which of these convergence criteria we use, we will need the same amount of communication. There are two obvious ways of implementing the convergence criteria. Either all the processes send their part of the relevant vectors to one process and this process test whether convergence has been reached. This process must then report back to the other processes to terminate if the convergence criteria has been met or to iterate again if not. The drawback of this is that it requires a lot of communication. First with several One-to-One communication statements and then a One-to-All statement later. This results in a lot of unused time as all the other processes have to wait for the one doing the convergence testing and a lot of temporary storage for the process doing the calculation. Pseudocode for this implementation can be found in Algorithm 3

Algorithm 3 Pseudocode for checking for convergence using only one process.

```
for each iteration k do
    Share needed values of u_prev
    Solve Jacobi's method
    Check for convergence
    if id == MASTER then                ▷ The MASTER process checks for convergence
        Receive data from all other processes
        Organize data in correct order
        Check if convergence has been reached
        if convergence reached then
            Tell other processes to return
            return
        else
            Tell other processes to iterate again
        end if
    else
        Send needed data to MASTER
        Wait for result
        Receive command from MASTER
        if convergence is reached then
            return
        else
            Start new iteration
        end if
    end if
end for
```

A better solution would be to have each process calculate their own internal maximum first and then simply compare this maximum between processes. Both our suggested convergence criteria have the possibility of being calculated locally by each process with the information that process controls. Then the local maxima can be compared between processes by using an `AllReduce` statement which returns the global maximum difference to each process so they themselves can test whether convergence has been reached. This is therefore a more efficient way to do things than the first suggestion. Using `AllReduce` requires significantly less communication than having to send all information to one process. There is no waiting time as all the processes can calculate their part of the convergence criteria at the same time and there is no need for temporary storage as the only value being sent around is the variable containing the maximum difference from each process. Pseudocode for checking for convergence using all processes and `AllReduce` can be found in Algorithm 4.

Algorithm 4 Pseudocode for checking for convergence using all processes and `MPI_AllReduce`.

for each iteration `k` **do**
  Share needed values of $\boldsymbol{u}_{prev}$
  Solve Jacobi's method
  Calculate local maximum difference
  Use `MPI_AllReduce` to find the global maximum difference and return it to all processes
  if global maximum difference $<$ tolerance **then** ▷ Convergence is reached
    return
  else ▷ Convergence not reached
    Iterate again
  end if
end for

6.5 Parallelization of the Conjugate-Gradient method

We will be using a pre-implemented version of the Conjugate-Gradient method (CG) to compare with how our classical iterative method is performing. As such we do not need to focus on the best way of implementing CG in parallel. It is however still interesting to briefly discuss the problems that arise when using CG to solve a linear problem in parallel as this will give us an idea of what to expect from the parallel results.

CG has the same problem as the other iterative methods when it comes to testing convergence, but this will require the same amount of communication regardless of iterative method so we can not use this to look at which method require the most communication.

To determine the expected performance of CG in parallel we have to look at the method itself. As shown in Section 4.9 there are three points in the Conjugate-Gradient method which will be the most obvious problems. The first is the matrix-vector product $\boldsymbol{t}_k = \boldsymbol{A}\boldsymbol{p}_k$. This needs to be calculated and distributed to all the processes in some way. To calculate it would require communication between process both in gathering up $\boldsymbol{p}_k$ and distributing the resulting $\boldsymbol{t}_k$ vector. This communication can be done effectively, but it will still require time to do. The second and third points are the calculation of the two parameters α_k and β_k as these are calculated by solving inner products. This can be done effectively, but will still require communication. These inner products can for instance be solved by each process solving their local part first and then adding these parts together and distribute the result using `AllReduce`.

Based on this we see that the Conjugate-Gradient method requires much more communication between processes than the classical iterative methods. Because of this it is worthwhile to compare Jacobi's method with CG using a parallel implementation even though CG was by far the fastest method when we solved the problem using serial computation. Jacobi does have a chance to catch up CG because it requires so much less communication between processes for each iteration. The disadvantage of Jacobi is that it generally requires many more iterations to converge. So even though it needs much less communication during each iteration the total communication might add up, possibly resulting CG in still being faster. This have to be tested. The results of these tests are given in Chapter 7.

6.6 Forward Euler

The Forward Euler method is very well suited for parallelization because it requires very little communication between processes. As it is not an iterative method there is no need for communication to check for convergence. This also ensures that there is only one round of communication for each time step. Like with Jacobi's method, the processes for Forward Euler only needs to communicate with its closest neighbours in each direction. If we are solving a 3D problem this would mean a maximum of six neighbours to communicate with.

For our parallel tests we will not investigate the performance of Forward Euler. We know from the tests done in Chai et al. (2013) that this dis-

cretization scheme will perform very well when implemented in parallel. As the communication demands on the solvers for the implicit methods is larger it is very important to figure out which solver is the most efficient when programming in parallel. We will therefore prioritize finding the best implicit solver. By implementing both Jacobi's method and CG, we will thereby see which of these iterative methods perform the best when implemented in parallel.

6.7 Methods implemented in parallel

As shown in the previous sections, we will implement Jacobi's method and compare it to the performance of the Conjugate-Gradient method. For the serial implementation CG was the only iterative method able to compete with Forward Euler. Therefore we will test if one of the classical iterative methods is able to compete with CG when used with multiple processes. We can assume that Jacobi's method will be better than CG when using a large number of processes as there is less requirement for communication between all processes for Jacobi compared to CG.

6.8 Implementing Jacobi's Method in C++

6.8.1 Adapting the Code

We adapted an existing code to solve our problem. This code was used for the numerical experiments in Hanslien et al. (2011). This code already used CG to solve a two-dimensional finite difference problem. The code was modified to support a three-dimensional finite difference problem. The initial conditions was changed so the code solved the same problem as we have looked at in the serial implementation. We also adapted the code to be able to use different iterative methods to solve the problem.

The packages used in this already existing code for handling matrices and vectors is the `Epetra` package from the Trilinos project (Trilinos 2015). By using the `AztecOO` library, another package from Trilinos, it was easy to solve the problem with CG as this was already an implemented option. You simply had to choose this as the solver and then start iterating. We wrote our own solver for Jacobi's method, since the `AztecOO` library do not have a solver for this iterative method.

The implementation of our methods in C++ can be found in Appendix C.

When adapting the code to fit our purposes there are two primary problems. The first is to modify the code so that we solve the same problem

as we did in the serial tests. To do this we have to implement support for solving a three-dimensional problem and change the initial and boundary condition to those of the problem we want to solve. The other problem is to implement Jacobi's method so that it is compatible with the original code.

6.8.2 Adding Support for 3D

Adding support for solving three-dimensional problems is easy. The code already has a variable called `nsd` which specify the number of dimensions. To be able to solve three-dimensional problems we simply have to add the correct logic for the z-axis and the z-coordinates to the two-dimensional logic already in the code. To not break support for two-dimensional problems we have added `if`-tests around the blocks of code which defines the coordinates. The problem is partitioned among the processes so we use the 2D logic when `nsd==2` and the 3D logic when `nsd==3`. All these changes where implemented in the file `RectDomain.cpp` which can be found in Appendix C.

6.8.3 Other Changes

Before we could start implementing Jacobi's method we had to make some other changes to `RectDomain.cpp`. First we had to add the option to define which solver we wanted to use from outside the program so we would not have to recompile the program every time we changed from using Jacobi to CG. Now the program takes an additional input argument, either "J" or "C" which indicates which of the two solvers we want to use.

We also had to modify the program so that we solved the problem that we wanted to solve. As the code originally solved a reaction-diffusion equation we had to remove the parts with the reaction equation from the code, so we could solve only the diffusion equation. We also had to change the initial condition to the one we wanted to use. The correct boundary conditions were already implemented. When we had made these changes our program was able to solve the same problem as the one we solved in serial by using the Conjugate-Gradient method.

6.8.4 Implementing Jacobi's Method

We chose to implement the solver for Jacobi's method as a class and not a function. This made it possible to use one function to set all the information that does not change for each iteration or time step and then have another function which performed the actual iteration. This gave the code a better structure than simply putting everything into a very long function. The code for our implementation of Jacobi's method is found in Appendix C. The relevant files are called `Jacobi.h` and `Jacobi.cpp`. Apart from the

constructor and deconstructor there were two functions we had to write in our Jacobi class. These were the functions to set the parameters and to solve Jacobi's method.

Jacobi::SetParameters

This function is used to set the general parameters needed to solve the problem using Jacobi's method. None of these parameters change as we iterate so each process only needs to call this function once. This function gets the information of which nodes this process shall solve the equations for so that each process keeps track of which part of the unit cube it is responsible for also inside this class. We also use this process to find which processes this process has as neighbours in each direction. We need to know this in order to exchange data with the correct processes when iterating over Jacobi's method later. We also expand the domain to include the ghost cells needed to store information received from other processes. This function also calculate how many nodes this process controls in each dimension and initializes all needed arrays, both the vectors needed to store $\boldsymbol{u}$ and $\boldsymbol{u}_{prev}$ and the arrays used for temporary storage when sending and receiving data. We also calculate the indices of the data we have to send to and receive from so that this can be collected and stored correctly in the ghost cells in $\boldsymbol{u}_{prev}$.

Jacobi::SolveJacobi

This function carries out the actual iterations of Jacobi's method and handles the communication between processes and solves the equations each process is responsible for. This function was the most challenging to get to work correctly in solving for each $\boldsymbol{u}_i$ and in the communication between the processes. This function does have room for improvements as there are large parts of the code that can be optimized further.

Most of the work that needs to be done before we start iterating has been done in the function `SetParameters`. In `SolveJacobi` we only have to set a few constant parameters and transfer the values from the initial guess vector, $\boldsymbol{u}_0$, to $\boldsymbol{u}_{prev}$. The reason for having to reset these constant parameters in this function and not simply using those generated in `SetParameters` for the whole class is that C++ does not, for obvious reasons, allow the declaration of `const int` at one place in the code, ie the header file, and setting the value at another place in the code. To keep these values declared as `const` we have to redeclare them in this function. Once this is done we can start the Jacobi iteration.

All of the challenges in writing this code arise inside the iteration loop. There are three things we have to consider here. We have to send and receive the needed information from the neighbouring processes, we have to solve each equation that this process is responsible for and we have to check

if our method has converged yet.

To communicate between processes we will be using the method of each process only exchanging information with its neighbours. As the values from the neighbouring processes have to be received before we can start using Jacobi's method, we start with solving how to send and receive these values. As the values we have to send are not stored sequentially in $\boldsymbol{u}_{prev}$ this require quite a lot of iteration. For each neighbour we have to iterate over $\boldsymbol{u}_{prev}$ and extract the values stored at the indices we set for that neighbour in `SetParameters`. These values are then stored in a vector we can send to the correct neighbour. To send and receive information from each neighbour we use the function `MPI_Sendrecv`. As we know that we should receive information from each process we have to send information to, this function is the most effective. The information received is then stored back into $\boldsymbol{u}_{prev}$ at the correct ghost cells. This way of getting and storing information can probably be improved as our current implementation requires a large amount of iteration over vectors, because the information we have to send is not stored sequentially in $\boldsymbol{u}_{prev}$. Storing the information sequentially will cause significant problems when we are going to use the vector to solve the equations with Jacobi's method. Pseudocode for how we share the needed information with neighbouring processes is found in Algorithm 5

Algorithm 5 Pseudocode for sending and receiving needed data from neighbouring processes.

```
for each iteration k do
    for each neighbouring process np do
        for all Values to send to np do
            Get value of u_prev at correct index and store in SendBuff
        end for
        Use MPI_SendRecv to send the values in SendBuff to np and store
the received values in RecvBuff
        for all Values received do
            Store all values in RecvBuff at the correct index in u_prev
        end for
    end for
    Solve Jacobi
    Check convergence
end for
```

When all needed information has been correctly sent and received we start iterating over the values of $\boldsymbol{u}$ that this process is responsible for calculating. Here we are slightly handicapped by the preexisting code. As we are using the `Epetra_CrsMatrix` structure to store our $\boldsymbol{A}$ matrix we have to spend

additional time extracting and iterating over information that might not otherwise have been necessary. We are using this structure simply because CG requires it. This structure is a disadvantage for Jacobi's method.

To solve the equations we use a triple `for`-loop to iterate over the nodes this process controls in each direction. We then use these iteration variables to calculate the index of the $\boldsymbol{u}$ value we are currently solving for. Before we can solve for this value we have to extract the relevant row from the $\boldsymbol{A}$ matrix and store the non-zero values of this row in a temporary vector for easy access. To calculate the sum $\sum_{j \neq i} \boldsymbol{A}_{i,j} \boldsymbol{u}_{prev}(j)$ we have to iterate over the values in this temporary storage. Here we run into another drawback of the `Epetra` structure. The values of the row of $\boldsymbol{A}$ are not stored in a predictable way. Therefore we have to use `if`-tests to determine which value from $\boldsymbol{A}$ should be multiplied by which value from $\boldsymbol{u}_{prev}$. This is not a very efficient way to do things as `if`-tests inside large `for`-loops are very time consuming. Once we have iterated over all the non-zero elements of the current row of $\boldsymbol{A}$ we have calculated the sum $s = \sum_{j \neq i} A_{i,j} \boldsymbol{u}_{prev}(j)$ and stored the diagonal element $\frac{1}{\boldsymbol{A}_{i,i}}$ in a. We are then able to calculate the current value i of $\boldsymbol{u}$ which we want to solve for by solving

$$\boldsymbol{u}(i) = \frac{1}{\boldsymbol{A}i, i} \left(\boldsymbol{b}_i - \sum_{j \neq i} \boldsymbol{A}_{i,j} \boldsymbol{u}_{prev}(j) \right) = (\boldsymbol{b}_i - s) * a.$$

Now all we have left to do is sort out how to check for convergence. As we discussed in Section 6.4.2, we are interested in ensuring that each process carries out as much of the calculation required for this as possible. We use

$$\max_i |\boldsymbol{u}(i) - \boldsymbol{u}_{prev}(i)| < \varepsilon,$$

as the convergence criteria as we can then avoid the matrix-vector product required by the other possible convergence criteria and therefore we will need less calculations for each process with regards to checking for convergence.

Each time we have solved for a value of $\boldsymbol{u}$ we check whether the difference $|\boldsymbol{u}(i) - \boldsymbol{u}_{prev}(i)|$ is larger than the difference for the previously calculated values of $\boldsymbol{u}$. If this difference is larger we store it in our `tdiff` parameter. Once we have solved for all the $\boldsymbol{u}$ values that this process controls, we will also have found the local maximum difference. If we are checking for convergence on this iteration we carry out a call to `MPI_AllReduce` which finds the largest global difference and distributes this to all processes. If this global difference is smaller than ε then the convergence criteria has been fulfilled and we return to the function which called this process. Iterations for the next time step is carried out if there are more, or the solution is saved if we have reached T_{end}. If convergence has not been reached we start over again on the next iteration. If we reach the maximum number of

iterations `max_it` we exit this function regardless of whether convergence has been reached or not. Ideally this should never happen. Pseudocode for this implementation of the convergence criteria can be found in Algorithm 6

Algorithm 6 Pseudocode for checking for convergence.

```
for each iteration k do
    for each neighbouring process np do
        Send and receive the needed data
    end for
    tdiff = 0.0
    for each u_i this process solves for do
        Solve Jacobi
        if |u(i) − u_prev(i)| > tdiff then
            tdiff = |u(i) − u_prev(i)|
        end if
    end for
    if we check convergence this iteration then
        MPI_AllReduce(tdiff,global_diff,MPI_DOUBLE,
MPI_MAX,MPI_COMM_WORLD)
        if global_diff ≤ ε then
            Store calculated values of u in return vector u_end
            return u_end
        end if
    end if
end for      ▷ Convergence not reached within the maximum number of
iterations
Store latest calculated values of u in return vector u_end
return u_end
```

There is one final thing we have to do before exiting the `SolveJacobi` function. Our $\boldsymbol{u}$ and $\boldsymbol{u}_{prev}$ vectors contain ghost cells for storing information from neighbouring processes. This means that we have to transfer the information into a vector $\boldsymbol{u}_{end}$ which has the expected size and contains only the elements this process has calculated and not any of the ghost cells.

6.8.5 Plotting the Solution

For debugging purposes it is useful to see the results from our solvers. Therefore we wrote a Python program to read the result files from our C++ program and plot this numerical solution as well as the analytical solution and the error between these two. As each process running the C++ program generates their own result file with their part of the result vector $\boldsymbol{u}$ this proved to be more challenging than expected. Not only did the Python

program have to read the binary file and plot the solution. We also had to consolidate all the result files and store the values of $\boldsymbol{u}$ in the correct order. Being able to plot the solution and view the results of our C++ program proved to be very useful when debugging our Jacobi solver as it made it much easier to find the possible sources of the error. This Python script is found in Appendix C, together with the C++ files, and is called `PlotRes.py`

Figure 6.2 shows an error plot generated by this python program. The solution plotted in this figure was calculated on a grid of $128 \times 128 \times 128$ elements, using 1024 processes. This plot is the parallel version of the error plot for Jacobi's method in Figure 5.9. We see that these to plots are very similar. The only difference is that the maximum value is slightly larger for the parallel version. This could be caused by the fact that the parallel version uses a different tolerance criteria for convergence. This criteria might be slightly more relaxed than the residual criteria used in the serial implementation. The x-axis and y-axis is different in the two plots. Python shows the solution over the z-plane of the unit cube. The MATLAB plots shows the nodes this plane was divided into instead.

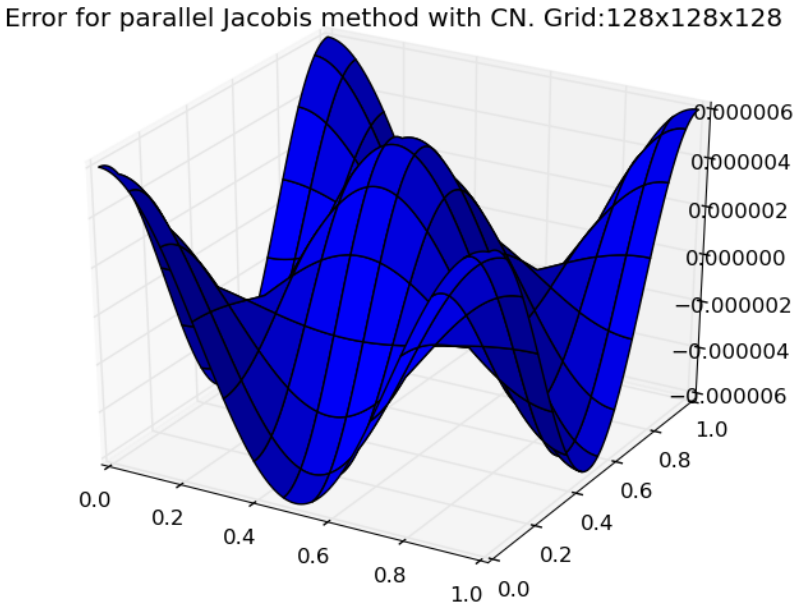


Figure 6.2: Error for Jacobi's method using Crank-Nicolson on a grid of $128 \times 128 \times 128$ *with 1024 processes*

Chapter 7

Parallel Results

7.1 Introduction

After implementing Jacobi's method as described in Section 6.8 we have compared the results with the results from the parallel Conjugate-Gradient method (CG) we discussed in Section 6.5. The results discussed in this chapter have been obtained by running both these methods on the supercomputer Vilje located at the Norwegian University of Science and Technology in Trondheim. This cluster has 2 eight-core processors per node and each processor has two threads so in total you can run 32 threads in parallel on each node. This is elaborated on by Eide and Jensen (2015).

The tolerance criteria for convergence is different in our parallel implementation compared to the serial results. As we discussed in Section 6.8.4, we used the difference between $\boldsymbol{u}$ and $\boldsymbol{u}_{prev}$ to test if the method has converged. To achieve the same accuracy as we did for the serial results, in Chapter 5, we have to use a stricter tolerance. For our parallel results we used the tolerance $\varepsilon = 10^{-8}$ as our convergence criteria.

7.2 Time Usage

7.2.1 Time used when solving with different number of processes

We know the main difference between Jacobi's method and CG from our serial tests and the theory behind the methods. For Jacobi's method, each iteration is relatively cheap, but it needs a lot of iterations to converge. For CG, each iterations are computationally expensive, but it only needs a few to converge. This is relevant when we look at the time used by each method for larger and larger number of processes. To obtain these results we have solved the same problem as we solved in serial on a grid of $128 \times 128 \times 128$ and $256 \times 256 \times 256$ elements using a parallel implementation of each method.

Grid: $128 \times 128 \times 128$

For this grid CG needs 7 iterations to converge while Jacobi's method uses 81 iterations.

p	1	2	4	8	16	32	64
t_{CG}	935.244	652.839	340.325	211.134	102.633	101.945	62.6297
t_J	10360.3	5230.07	2661.88	1397.33	695.524	703.58	356.926
p	128	256	512	1024	2048	4096	
t_{CG}	49.8256	65.9159	128.883	232.937	473.204	938.319	
t_J	193.693	105.853	48.6887	49.3825	54.6842	63.6612	

Table 7.1: Total time used when solving the problem on the grid $128 \times 128 \times 128$ *for different number of processes, where* p *is the number of processes,* t_{CG} *is the time used by CG and* t_J *is the time used by Jacobi's method.*

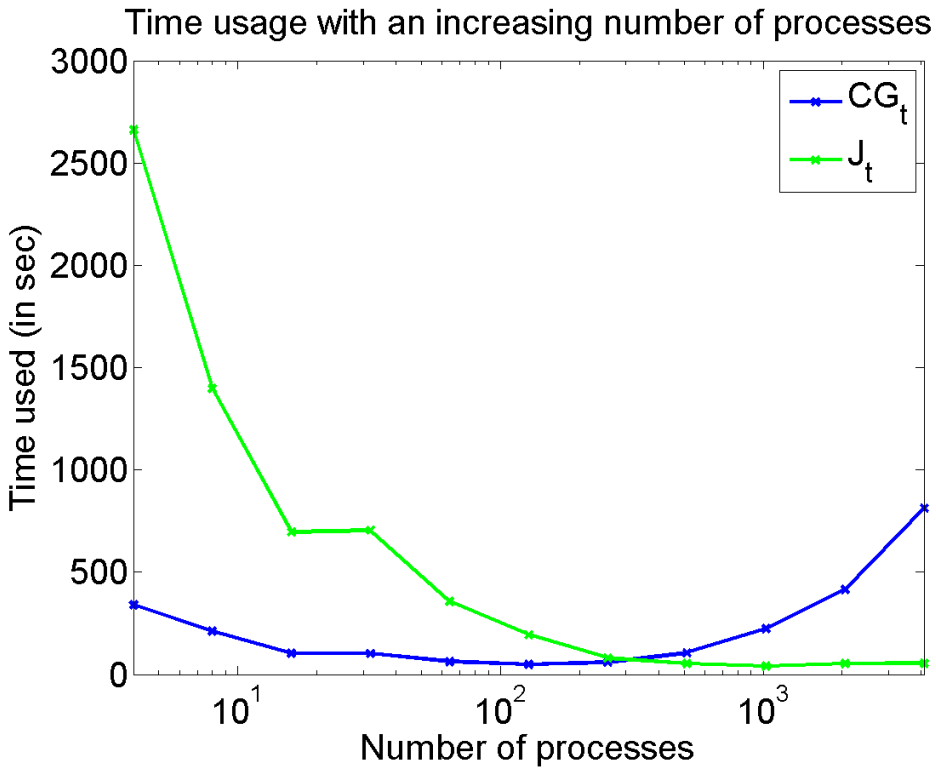


Figure 7.1: Visual representation of the data in Table 7.1. The total time used when solving the problem on the grid $128 \times 128 \times 128$ *for an increasing number of processes.*

Table 7.1 and Figure 7.1 shows the results for the grid $128 \times 128 \times 128$. Figure 7.1 does not include the results when using one or two processes.

As expected CG is still much better than Jacobi's method with a small number of processes. This is because for a small number of processes the computation time each process needs still overshadows the time required for communication between processes. So when solving Jacobi's method each process needs to solve the linear equations 81 times for each time step in order to converge. And because each process still controls a relatively large number of the equations needed to be solved, Jacobi's method uses much more time than the 7 iterations per time step that CG needs. As the number of processes increases, we see that both methods scale very well at the start. For very few processes we see that for Jacobi's method the time used is approximately halved each time the number of processes is doubled. For CG the time used is reduced by between a third and half each time we double the number of processes. But this changes for CG when we get to 128 processes. This seems to be the optimal number of processes for solving the problem using CG on this grid as this is the shortest time CG uses on solving the problem. When we increase the number of processes beyond 128 CG starts using more time instead of being more effective, as shown in Figure 7.1 and Table 7.1. This is because CG is so communication heavy. After 128 processes the increase in time needed to communicate all needed data between the processes becomes larger than the time saved from each process solving a smaller number of equations. We see for CG that from 256 processes and up the time is approximately doubled when the number of processes is doubled.

For Jacobi's method, Figure 7.1 shows a different trend. It starts out using much more time than CG for very few processes. The reason for this is the fact that it needs so many more iterations to converge, and therefore more calculation has to be done. Because Jacobi's method starts with using so much more time than CG, it requires a significant increase in processes used for this method to get down to the same level of time used as CG. The results shows that Jacobi's method has to use more than 256 processes to use less than 100 seconds. We see that the method uses the least time for 512 processes. It actually uses less time than CG did at its minimum time used. With more than 512 processes the time usage starts increasing again, but unlike CG the increase in time use is very small. So from these results it would appear that CG is much better for a small number of processes while Jacobi becomes more and more viable the more processes used. We also see that Jacobi's method does not only catch up to CG as the number of processes increases, it is actually much better than CG when using a very large number of processes to solve the problem. So it would seem that CG is the best choice when we only have access to a small number of processes to solve the problem in parallel, while Jacobi's method should be used when we can use a large number of parallel processes.

Table 7.1 also show some unexpected results when we go from 16 to 32

processes. For both methods the time used is not decreasing as expected based on the time used for the previous numbers of processes. Both methods use approximately the same time when solving with 16 processes and 32 processes. We also see that when we go from 32 to 64 processes the expected trend resumes again. We are not quite sure why this happens. It might be related to the architecture of the computer we are running our program on as this cluster has cpus with 16 processors, but as each processor has 2 threads it would have been more likely that we had noticed something odd when going from 32 to 64 processes and not from 16 to 32.

Another thing to note about Table 7.1 is the time used when solving with one process. We see that the time used here is much larger than the time usage we measured in our serial tests discussed in Section 5.3.7. This is most likely caused by Matlab as we do not have complete control over how the optimized matrix-vector operations take advantage of the computer it is running on. Monitoring the cpu usage when solving our problem in Matlab, it appears that Matlab is using several of our available processors so it is quite possible that there is some parallelization going on behind the scenes which makes the serial Matlab code run faster than the "serial" version of our parallel implementation. This could also be part of the reason of why the component forms were so much slower than the matrix forms when we tested our iterative methods in Matlab.

Grid: $256 \times 256 \times 256$

For this grid Jacobi's method needs 250 iterations to converge while CG only needs 8.

p	32	64	128	256
t_{CG}	930.187	477.568	307.559	225.716
t_J	17473.7	8906.91	4670.61	2401.53
p	512	1024	2048	4096
t_{CG}	273.306	511.799	1022.36	2172.90
t_J	1322.46	775.426	492.652	366.107

Table 7.2: Total time used when solving the problem on the grid $256 \times 256 \times 256$ *for different number of processes, where p is the number of processes,* t_{CG} *is the time used by CG and* t_J *is the time used by Jacobi's method.*

Table 7.2 and Figure 7.2 show the time used for various numbers of processes on the grid $256 \times 256 \times 256$. Figure 7.2 does not include the results for 32 processes.

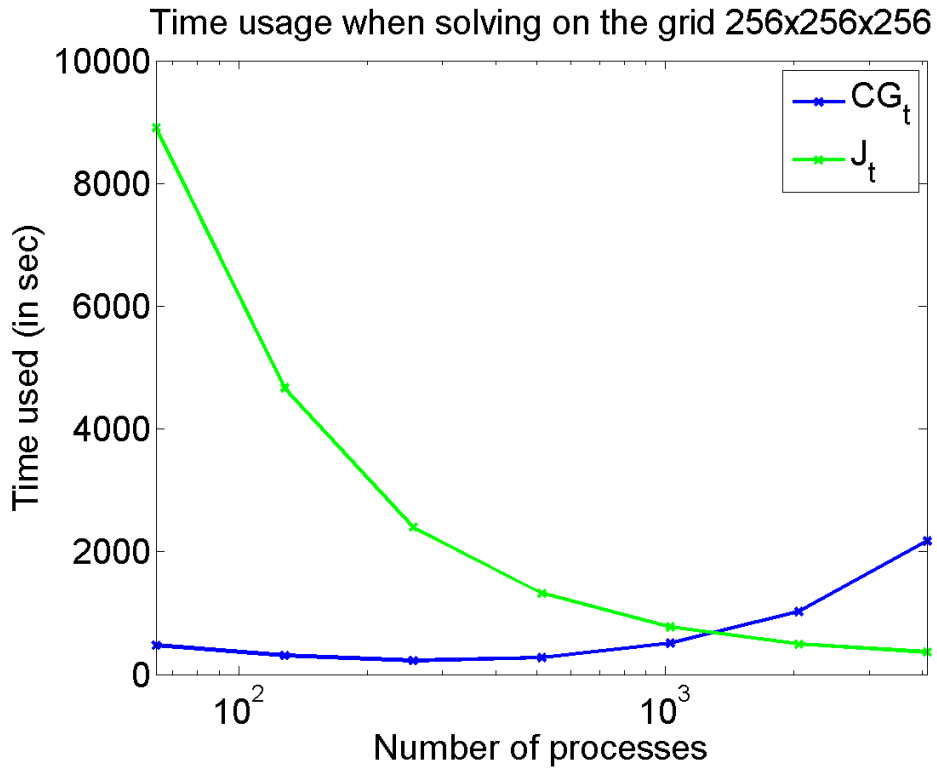


Figure 7.2: Visual representation of the data in Table 7.2. The total time used when solving the problem on the grid $256 \times 256 \times 256$ *for an increasing number of processes.*

The total time used has changed for this grid compared to $128 \times 128 \times 128$, but the trend is the same for both grids. There are however some relevant differences to discuss. When we go from solving our problem on a grid of $128 \times 128 \times 128$ to $256 \times 256 \times 256$, there is a significant increase in the number of equations that must be solved, from 2146689 to 16974593 unknowns. This results in some expected differences in the time used for the two grids. As there are so many more equations to solve, CG is better than Jacobi's method for a larger number of processes as it takes much longer before the increase in time used to communicate between processes is larger than the time saved by each process solving fewer equations. Here, as well, will Jacobi's method eventually catch up to CG. It just takes a larger number of processes to do it.

The results also show that the general time use has increased a lot. The minimum time used has gone from under 50 seconds to over 200 seconds. Two factors contribute to this. One is the fact that we have significantly increased the problem size. This increases the computations required by each process for each iteration so each process needs much more time to finish solving their part of the problem. The other factor is the increase in data that each process needs to share. As the problem size has become

larger the amount of data that needs to be sent and received by each process has also significantly increased and this also requires more time to complete.

For Jacobi's method there is an additional factor which increase the time used, and that is the increase in iterations needed for convergence. Going from 81 iterations per time step to 250 increases both the computation and communication needed for each time step. Therefore the method uses more time to solve the problem for this grid.

We also see that for this grid size the minimum time used by Jacobi's method is still larger than the minimum time used by CG. This is different from what the results in Table 7.1, where the minimum value for Jacobi's method actually was smaller than the minimum value for CG. One reasons for Jacobi not getting down to the minimum value of CG for this grid might be because the difference in iterations required for each method has become so much larger. It is also possible that Jacobi's method can get down to the minimum value of CG if we use an even larger number of processes.

Another interesting difference between Table 7.1 and Table 7.2 is the behaviour of Jacobi's method for the largest numbers of processes. For the grid $128 \times 128 \times 128$ the time used starts slightly increasing again when we use more than 512 processes. For the grid $256 \times 256 \times 256$ the time used is still decreasing when we go from using 2048 processes to 4096 processes.

7.2.2 Time used for different grids

N	8	16	32	64	128	256
k_{CG}	2	3	4	5	7	8
t_{CG}	0.333947	0.357824	1.36639	10.3035	102.54	941.393
k_J	3	5	10	27	81	250
t_J	0.0681301	0.228161	1.78223	29.7046	695.567	17764.3

Table 7.3: Total time used when solving with 16 processors on various grids. k_{CG} and k_J are the number of iterations needed for convergence, while t_{CG} and t_J is the time used to solve the problem. N is the number of elements in each direction.

N	16	32	64	128	256
k_{CG}	3	4	5	7	8
t_{CG}	34.2429	31.5203	54.6616	62.8039	224.698
k_J	5	10	27	81	250
t_J	1.12562	2.10715	9.53529	107.605	2424.00

Table 7.4: Total time used when solving with 256 processors on various grids. N, k_{CG}, t_{CG}, k_J and t_J are the same as in Table 7.3

N	32	64	128	256
k_{CG}	4	5	7	8
t_{CG}	357.052	241.918	233.654	501.469
k_J	10?	27	81	250
t_J	8.60766	11.3558	50.7144	780.825

Table 7.5: Total time used when solving with 1024 processors on various grids. N, k_{CG}, t_{CG}, k_J and t_J are the same as in Table 7.3 and Table 7.4

To see how our parallel implementation behaves when we solve for larger and larger grids we have tested the time used on different grids with three different numbers of processes. In Table 7.3 we have solved the problem on the same grids as the serial results using 16 processes, in Table 7.4 we have done the same using 256 processes and in Table 7.5 we used 1024 processes to see how the methods behave. We have also solved for the grid $256 \times 256 \times 256$ to see the behaviour when the problem size increases even more. The reason some of the results differ slightly in this section compared to the previous section is simply that the measured time used is from different runs of the same problem.

The results are largely as expected. For the coarser grids, the number of iterations needed for convergence is almost equal for CG and Jacobi's method and then they use almost the same amount of time. Jacobi's method is slightly faster as each iteration is more computationally heavy for CG. However when the iteration numbers drastically increases for Jacobi's method we see that the time used becomes much larger than CG. But if we look at the time increase compared to how many iterations each method needs to converge we see that this is much larger for CG. This is particularly true for when we use a large number of processes as CG requires much more communication between processes. When we increase the number of processes, the results shows that Jacobi's method gets better compared to CG the more processes we use. This is the same behaviour as the results in the previous section showed.

If we compare these results to the serial results in Chapter 5 we see that the methods use fewer iterations to converge in the parallel implementation. The reason for this is that we have used a smaller Δt. This results in fewer iterations to reach convergence for each time step, but we need to solve the problem for more time steps. The reason for using a smaller Δt is two fold. The code we adapted had Δt as an input parameter instead of it being dependent on the space discretization like we did in Matlab. The other reason is that using a Δt as close to the serial Δt as possible resulted in some trouble with reporting the solution at $t = T_{end}$ as the program did not hit exactly T_{end} when using this Δt. The Δt used is still much larger than the Δt required for a stable explicit solver like Forward Euler. For

the serial implementation we used a Δt dependent on h, like we discussed in Section 5.3, which results in a larger Δt for coarser grids. For the finest grid used in the serial results, $128 \times 128 \times 128$, this gave a $\Delta t \approx 0.032$. For our parallel tests we have used a set value of Δt regardless of the grid size. This value is $\Delta t = 0.025$.

We use a difference criteria for convergence for our parallel tests which, even with a tolerance of 10^{-8}, is slightly more relaxed than the criteria used in the serial tests. This is seen when comparing the error plot in Figure 6.2 with the error plot for Jacobi's method in Figure 5.9. We see that the error for the parallel implementation is slightly larger than for the serial implantation. This could also contribute to the reduction in the iteration numbers.

7.2.3 Partitioning

An interesting part of solving problems in parallel is how to divide the problem between the processes. We are solving our problem on the unit cube. The cube is divided into parts and we assign each part to a process so each process knows which nodes and therefore which equations they are responsible for. How we do this decides what sort of communication we need between processes. If we partition only along one axis each process will need to communicate with fewer processes, but they will have to send and receive much larger data sets. While if we try to make each partition as equal in each direction as possible each process has to communicate with more processes as they will have more neighbours. But the amount of data they have to send to and receive from each process will be much smaller.

We have done some tests on what sort of partitions which suits our problem the best. There is very little difference in time used between partitioning as equal as possible along all three axes and partitioning as equal as possible along two axes, while using 1 partition along the last axis. However the time difference becomes very large between this sort of partitioning and using all partitions along one axis. For instance partitioning the grid $128 \times 128 \times 128$ on 128 processes using the partition $[128, 1, 1]$ or $[1, 1, 128]$ took almost twice as much time as using the partition $[16, 8, 1]$. Using the partition $[64, 2, 1]$ gave almost the same results as $[128, 1, 1]$, while the partition $[8, 4, 4]$ gave approximately the same results as $[16, 8, 1]$. From our tests it does not appear to make a significant difference which way we partition the cube. The partition $[16, 8, 1]$ uses the same amount of time as $[1, 8, 16]$.

7.2.4 Convergence Testing

As we discussed in Section 6.4.2, the convergence testing may slow down the computation in our program. Because of this, we implemented the option

to decide how often we want to check for convergence, instead of having to check each iteration. We implemented the convergence criteria so that it allows each process to calculate their local maximum difference. Then the communication is reduced to a simple `AllReduce` statement which compares and finds the global maximum difference and distributes this value to each process.

When testing this functionality we found that it was faster to check for convergence each time. This was because it took more time to solve the problem for the additional iterations needed before we checked for convergence again, than it took using the `AllReduce` communication statement each iteration. For instance if you check for convergence every 10 iterations, you can in the worst case scenario end up with iterating an additional nine times from when convergence was reached and until we test for convergence again. Thus our tests showed that the convergence test is very efficient. As a consequence these extra iterations cost us so much extra computation time that is was more efficient to test for convergence at each iteration.

It is likely that the convergence test would not be as efficient if we had used the other suggested convergence criteria, because implementation of this criteria would need the calculation of a matrix-vector product each time we checked for convergence.

Chapter 8

Conclusions and Further Work

8.1 Conclusion

From the tests we have done in this thesis there are two primary results that are achieved. The serial tests shows that all methods behave as expected based on the theory and that the performance of the classical iterative methods are not able to compete with the Forward Euler method for our test problem. The only method which is faster than Forward Euler in our serial tests is the Conjugate-Gradient method (CG) which is computationally expensive, but fast converging. The classical iterative methods are far behind CG when we use a serial implementation. However, the results for the parallel tests are quite different. We implemented Jacobi's method and the results showed that this method is able to outperform CG in certain cases. This is a much better result for our classical iterative methods than what we got from the serial tests. This indicates that it is worthwhile to continue studying the performance of classical iterative methods with parallel implementation.

8.1.1 Results from the Serial Tests

For our serial results we have been looking at several different and interesting parts that affect the performance of our iterative methods.

We have looked at the performance of all the methods compared to each other when it comes to time used, convergence rate and resulting errors. The results of these tests where largely as expected, based on the theory. Gauss-Seidel's method performed slightly better than Jacobi's method, but they were both beaten by Forward Euler and the Conjugate-Gradient method when it came to time used. This was not unexpected, but a more surprising observation is the bad performance of Richardson's method. This is most likely caused by the fact that our matrix $\boldsymbol{A}$ fit very badly for this method because of its eigenvalues, as we discussed in Section 5.3.4. Richardson's method, unlike the other iterative methods, depends on these values to

guarantee convergence.

In addition to looking at how the solvers performed compared to each other we also looked at how the two implicit discretization schemes performed. When comparing Backward Euler to Forward Euler, we see that Backward Euler actually gives larger errors than Forward Euler. Since both these schemes are first order schemes, they have the same convergence rate. Therefore Backward Euler would need Δt dependent on h^2 to be as accurate as Forward Euler, and that explains the larger errors. Crank-Nicolson however gives much better results than Forward Euler. This is because Crank-Nicolson is a second order scheme so it is able to achieve the same rate of convergence, with Δt dependent on h, as Forward Euler has with Δt depending on h^2. Because of this Crank-Nicolson has a huge advantage over Backward Euler when we are comparing the results with the results from Forward Euler. The only advantage of Backward Euler is that it is unconditionally stable.

We also discovered interesting results when it came to the parameters for our solvers. The first of which is the tolerance for convergence. After extensive testing we found that the tolerance $\varepsilon = 10^{-7}$ was an optimal convergence criteria. Any more relaxed than this and the convergence rate towards the analytical solution would significantly decrease since the error accumulates from time step to time step. Using a smaller tolerance did not yield any noticeably smaller errors when comparing to the analytical solution, so it simply resulted in more time used on iteration with no accuracy gained in the solution, as we discussed in Section 5.3.

The other interesting parameter is Δt. While Forward Euler has very strict restrictions on Δt in order to ensure convergence, there is no such restrictions on the implicit methods. As such we tested with various values of Δt and unsurprisingly found that the iterative methods gave significantly improved results compared to Forward Euler when using a much larger Δt. But we also found that Δt could not be too large or we would lose too much accuracy compared to the analytical solution. As we discussed in Section 5.3, we found that the ideal values for the iterative methods were

$$\Delta t = \frac{C\beta h}{6}$$

with the diffusion coefficient $\beta = \frac{1}{100}$ for our test problem. Through testing we found that $C = 0.25$ was the optimal value for Crank-Nicolson when balancing efficiency and accuracy, and $C = 0.1$ was the optimal value when using Backward Euler. However with this time step the accuracy using Backward Euler is not nearly as good as Crank-Nicolson.

We also tested with different choices of initial guess vectors and found that

it became more important to make a good initial guess the more relaxed the tolerance for convergence was. We got the best results when using the $\boldsymbol{u}$ we found on the previous time step as the initial guess vector, $\boldsymbol{u}_0$, for the current iteration.

We also found that using a less numerically optimal initial condition changed the number of iterations required for each method to converge, but did not change the results of the methods compared to each other. The increase in iterations and time use was approximately equal for all methods.

Based on these results we concluded that we would have to use Crank-Nicolson as the time discretization scheme as it performed much better than Backward Euler, and that there would be no point in studying Richardson's method further as its results were so much worse than the other methods.

8.1.2 Results from the Parallel Tests

Based on the theory of parallelization and the communication required for each method we discarded Gauss-Seidel's method and proceeded with implementing only Jacobi's method of the classical iterative methods.

The results from our parallel tests are very good with regards to Jacobi's method. While CG still performs better for a smaller number of processes, Jacobi's method not only catches up to CG, but actually is faster than CG for a large number of processes. When solving on the grid $128 \times 128 \times 128$ we actually found that the minimum time used for Jacobi's method was smaller than the minimum time used by CG. The only disadvantage of Jacobi's method compared to CG seems to be that CG will still be more effective when the system of linear equations is very large. We see hints of this when looking at the grid $256 \times 256 \times 256$, when Jacobi's method is better than CG for the largest numbers of processes, but the minimum time used is still larger than the minimum time used by CG. In this comparison it should however be remembered that we will most likely get a much better performance from Jacobi's method if the code is fully optimized like we discuss below. It should also be noted that it is much more important when using CG to not use too many processes because communication is so expensive for this method. You have to find the perfect balance between not using too few so you get the advantages of parallelization and you can not use to many because then there is too much communication. This is much less critical for Jacobi's method, as it gives much better results for a very large number of processes.

8.2 Further Work

Even though the results from our serial tests were negative with regards to the classical iterative methods, the results from our parallel tests were so positive that they warrant further study.

8.2.1 Optimization of the Jacobi Code

The current implementation of the Jacobi code has significant room for improvement and still gives results which are competitive with the Conjugate-Gradient method when using a large number of processes. As we can expect the time used by Jacobi's method to significantly decrease when the code is optimized it will be interesting to see how it then performs compared to CG.

There are several things in our implementation of Jacobi's method which should be possible to improve. The first thing is the way we store $\boldsymbol{A}$. Now we are using the same method for creating and storing the matrix as CG uses, by using the `Epetra` package. This way is very well suited for CG as the `AztecOO` package where CG is implemented is written especially for efficiently using the `Epetra` vector and matrix structures. Jacobi's method is not specifically designed for efficiently using this structure. Therefore it is very likely that we are actually loosing time when we use this structure instead of using something that is better suited. The largest problem with using the `Epetra` structure for our $\boldsymbol{A}$-matrix is when we have to get the values of each row of this matrix when solving each equation in the Jacobi method. We have to get the values from the row we are solving for and store them in a temporary vector. As the values are not consistently ordered when we get them from the matrix we also have to use an `if-else`-block to get the values multiplied with the correct values of $\boldsymbol{u}_{prev}$ so the correct sum is calculated. This is a major disadvantage as one would like to avoid time-consuming operations like `if-else`-statements inside `for`-loops. This will also be an increasingly time consuming structure the more equations each method has to solve for each iteration. Thus will this becomes more and more inefficient the more refined the grid becomes. Improving this to avoid the `if-else`-block inside the `for`-loop will most likely be the most significant improvement for the code.

Another part of the code that can possibly be made more efficient is obtaining the values to send to the neighbouring processes and storing the values received from these same processes.

8.2.2 Other tests

We have done extensive testing for our serial implementation, but for the parallel testing we have not done nearly as much testing. This is partly be-

cause the serial tests indicated that some of our methods where not worthwhile to implement in parallel. Our results show that Jacobi's method can compete with CG when we use a large number of processes and when this code is optimized it will most likely have a much better performance compared to CG also for a smaller number of processes. What would be interesting to look at would be the performance of these two methods compared to Forward Euler. We know from our serial tests that CG, but not Jacobi, was performing better than Forward Euler and it would be interesting to test whether they both are able to compete with Forward Euler in parallel. When using parallel programming Jacobi's method is as efficient as CG, but CG is likely less competitive with Forward Euler as Forward Euler is very well suited for parallel implementation.

We know from our serial tests that having a smaller Δt reduced the number of iterations required for convergence for each time step. However, as it increases the number of time steps this might not necessarily give faster solutions. As we know from our parallel tests and from the theory, Jacobi's method is faster than CG in each iteration, but it looses time compared to CG because it needs so many iterations to converge. Because of this Jacobi's method would most likely give better results compared to CG with a smaller time step, but if the time step becomes too small both these methods will be outperformed by Forward Euler. Using as large a time step as possible would give these methods better performance compared to Forward Euler, but Jacobi could loose ground compared to CG because the number of iterations needed for each time step would increase as Δt increases. This shows that the effect of Δt is an interesting topic for further work.

It would also be interesting to test how using a preconditioner will affect the performance of CG compared to Jacobi's method. This might further improve the convergence rate of this method. However, it might also increase the communication required.

Appendix A

Implementation of the Iterative Methods in MATLAB

matlab/R_comp.m

```
function [u,k,err] = R_comp(A,b,u,alpha,K,eps)
%Implementation of the component form of Richardson iteration

n = size(A,2);
u_prev = u;
err = zeros(K,1);

for k=1:K
    for i=1:n
        r = b(i) - A(i,:)*u_prev;
        u(i) = u_prev(i) + alpha*r;
    end

    u_prev = u;
    r = b - A*u;
    err(k) = max(abs(r));
    if(err(k) < eps)
        err = err(1:k);
        return
    end
end
k = K+1;
end
```

matlab/R_matrix.m

```
function [u,k,err] = R_matrix(A,b,u,alpha,K,eps)
%Implementation of the matrix form of Richardson Iteration.
```

```
%Iterates until a tolerance eps or a maximum number of
    iterations K is reached

u_prev = u;
err = zeros(K,1);

for k=1:K
    u = u_prev+alpha*(b-A*u_prev);

    r = b-A*u;
    err(k) = max(abs(r));
    if (err(k) <eps)
        err = err(1:k);
        return
    end
    u_prev = u;
end
k = K+1;
end
```

matlab/J_comp.m

```
function [u,k,err] = J_comp(A,b,u,K,eps)
%Implementation of the component form of the Jacobi
    method

n = size(A,2);
u_prev = u;
err = zeros(K,1);

for k=1:K
    for i=1:n
        r = A(i,:)*u_prev - A(i,i)*u_prev(i);
        u(i) = (1/A(i,i))*(b(i)-r);
    end
    u_prev = u;

    res = b - A*u;
    err(k) = max(abs(res));
    if (err(k) < eps)
        err = err(1:k);
        return
    end
end
k = K+1;
end
```

matlab/J_matrix.m

```
function [u,k,err] = J_matrix(A,b,u,K,eps)
%Implementation of the matrix form of the Jacobi
    method

u_prev = u;
err = zeros(K,1);

d = diag(A);
D = diag(d);
R = A - D;

for k=1:K
    u = D\(b-R*u_prev);

    r = b - A*u;
    err(k) = max(abs(r));
    if (err(k) < eps)
        err = err(1:k);
        return
    end
    u_prev = u;
end
k = K+1;
end
```

matlab/GS_comp.m

```
function [u,k,err] = GS_comp(A,b,u,K,eps)
%Gauss-Seidel component form

n = size(A,2);
u_prev = u;
err = zeros(K,1);

for k=1:K
    for i=1:n
        s1 = A(i,1:i-1)*u(1:i-1);
        s2 = A(i,i+1:n)*u_prev(i+1:n);
        u(i) = (1/A(i,i))*(b(i)-s1-s2);
    end
    u_prev = u;

    r = b - A*u;
    err(k) = max(abs(r));
    if (err(k) < eps)
```

```
19          err = err(1:k);
20          return
21      end
22  end
23  k = K+1;
24  end
```

matlab/GS_matrix.m

```
1  function [u,k,err] = GS_matrix(A,b,u,K,eps)
2  %Matrix form of Gauss-Seidel iteration
3
4  u_prev = u;
5  err = zeros(K,1);
6
7  L = tril(A);
8  U = triu(A,1);
9
10 for k=1:K
11      u = L\(b-U*u_prev);
12
13      r = b - A*u;
14      err(k) = max(abs(r));
15      if (err(k) < eps)
16          err = err(1:k);
17          return
18      end
19      u_prev = u;
20 end
21 k = K+1;
22 end
```

matlab/cg.m

```
1  function [u,k] = cg(A, b, u, K, eps)
2  %Conjugate gradient method for solving Ax=b when A is
       symmetric positive definite
3
4  r = b - A*u;
5  p = r;
6  rho0 = b'*b;
7  rho = r'*r;
8
9  for k=0:K
10     if sqrt(rho/rho0) <=eps
11         return
12     end
```

```
13      t = A*p;
14      alpha = rho/(p'*t);
15      u = u + alpha*p;
16      r = r - alpha*t;
17      rhos = rho;
18      rho = r'*r;
19      p = r + (rho/rhos)*p;
20
21  end
22  end
```

matlab/J_wm.m

```
1   function [u,k,err] =J_wm(A,b,w,u,K,eps)
2   %Weighted Jacobi's method
3
4   u_prev = u;
5   err = zeros(K,1);
6
7   d = diag(A);
8   D = diag(d);
9   R = A - D;
10
11  for k = 1:K
12      u = D\(b-R*u_prev)*w + (1-w)*u_prev;
13
14      r = b - A*u;
15      err(k) = max(abs(r));
16      if (err(k) < eps)
17          err = err(1:k);
18          return
19      end
20      u_prev = u;
21  end
22  k = K+1;
23  end
```

Appendix B

Solver for the Diffusion Equation

matlab/laplacian.m

```
function A = laplacian(n, sym)

e = ones(n,1);
D1 = spdiags([-e 2*e -e], [-1 0 1], n, n);

% Set Neumann boundary conditions
if not(sym)
  % 2. order, breaks symmetry
  D1(1,2) = -2;
  D1(n,n-1) = -2;
else
  % Only 1. order, keeps symmetry
  D1(1,1) = 1;
  D1(n,n) = 1;
end

% Form A using tensor products of lower dimensional
    Laplacians
I = speye(n);
A = kron(I, kron(I, D1)) + kron(I, kron(D1, I))+ kron(
    kron(D1,I),I);
```

matlab/run.m

```
for tf = [0,1]
Lx = 1;
Ly = 1;
Lz = 1;

set(0,'DefaultAxesFontSize', 20)
set(0,'DefaultTextFontSize', 20)
set(0,'defaultlinelinewidth',2)
```

```
9
10 runCN = tf;
11 %runCN = 0; % 1 if running Crank-Nicolson
12 runIE = not(runCN); % 1 if running Backward Euler
13 runCG = 0; % 1 if running CG
14 comp = 0; %1 if running the component forms, 0 if not
15 runNorms = 0;
16 runSign = 0; %Run with Sign func as init condition
17 Gelim = 0; %1 if we want to solve IE by Gauss elim.
18 plotting = 0; % 1 if we want to plot convergence rates
       during the run
19 plot_errs = 1;
20 TN = 5;
21 K_it = 5000;
22 err_tol = 10^-7;
23 w = 2/3;
24
25 if runCN
26     C = 0.25
27 else
28     C = 0.1
29 end
30
31 Q = 7
32
33 if (Q <= 6)
34     runR  = 1;
35 else
36     runR = 0;
37 end
38
39 %Code for initializing all vectors used to store
       results removed. All these vectors were initialized
       as <res_vec> = zeros(1,Q);
40
41 tic
42 for q = 1:Q
43
44     h = 0.5^q
45     nx = (Lx/h) + 1;
46     ny = (Ly/h) + 1;
47     nz = (Lz/h) + 1;
48     n = nx*ny*nz
49     [x,y,z] = meshgrid(0:h:Lx,0:h:Ly,0:h:Lz);
50     if runSign
```

```
51          v0 = sign(x-0.5);
52      else
53          v0 = cos(2*pi*x).*cos(2*pi*y).*cos(2*pi*z);
54      end
55
56      v = v0(:);
57      clear x; clear y; clear z; %save some memory
58
59      beta = 100;
60
61      A = (laplacian(nx,runCG)/beta)/h^2;
62
63      dte = beta*h^2/6; % time step restriction for exp.
            Euler.
64      dt = C*beta*h/6;
65
66      if (runR)
67          eigvals = tic;
68          lmax = eigs(A,2);
69          lmax = lmax(1)
70          lmin = eigs(A,2,'sr');
71          lmin = lmin(2)
72          alpha = 2/(lmax+lmin)
73          toc(eigvals)
74          lx(q) = lmax;
75          lm(q) = lmin;
76      end
77
78      T_end = TN;
79      n_steps = ceil(T_end/dte) + 1; % + 1 to be certain
80      te = linspace(0, T_end, n_steps);
81      dte = te(2)-te(1);
82
83      n_steps = ceil(T_end/dt) + 1; % + 1 to be certain
84      t = linspace(0, T_end, n_steps);
85      dt = t(2)-t(1);
86
87      if runIE
88          M = speye(n) + dt*A; % Matrix for Imp. Euler
89      else
90          Mcn = speye(n) + 0.5*dt*A; % Matrix for Crank-
                Nicolson
91          Mcn_p = speye(n) - 0.5*dt*A; % Matrix for CN
92      end
93
```

```
94      if runNorms
95          if (q <= 4)
96              if runCN
97                  d = diag(Mcn);
98                  D = diag(d);
99                  R = Mcn - D;
100                 Gjm = D\R;
101                 clear d; clear D; clear R;
102                 Gnorm_jm(q) = normest(Gjm);
103                 clear Gjm;
104
105                 L = tril(Mcn);
106                 U = triu(Mcn,1);
107                 Ggsm = L\U;
108                 clear L; clear U;
109                 Gnorm_gsm(q) = normest(Ggsm);
110                 clear Ggsm;
111             else
112                 d = diag(M);
113                 D = diag(d);
114                 R = M - D;
115                 Gjm = D\R;
116                 clear d; clear D; clear R;
117                 Gnorm_jm(q) = normest(Gjm);
118                 clear Gjm;
119
120                 L = tril(M);
121                 U = triu(M,1);
122                 Ggsm = L\U;
123                 clear L; clear U;
124                 Gnorm_gsm(q) = normest(Ggsm);
125                 clear Ggsm;
126             end
127         end
128     end
129
130     eps = err_tol;
131     K = K_it;
132
133     if comp
134         v_rc = v;
135         v_jc = v;
136         v_gsc = v;
137     end
138
```

```
139     if (runR)
140         v_rm = v;
141     end
142     v_jm = v;
143     v_gsm = v;
144     if runCG
145         v_cg = v;
146     end
147     v_bicg = v;
148     v_jw = v;
149
150     if runIE
151         if Gelim
152             v_ie = v;
153         end
154     else
155         v_cn = v;
156     end
157     v_ee = v;
158
159     tsee = tic;
160     for i = 1:(length(te)-1)
161         v_ee = v_ee  - dte*A*v_ee;
162     end
163     tee = toc(tsee);
164
165     loop = tic;
166     if runCN
167         if comp
168             if (runR)
169                 tsrc = tic;
170                 for i = 1:(length(t)-1)
171                     b = Mcn_p*v_rc;
172                     [v_rc, k_rc, e_rc] = R_comp(Mcn,b,
                            v_rc,alpha,K,eps);
173                 end
174                 trc = toc(tsrc);
175             end
176             tsjc = tic;
177             for i = 1:(length(t)-1)
178                 b = Mcn_p*v_jc;
179                 [v_jc, k_jc, e_jc] = J_comp(Mcn,b,v_jc,
                        K,eps);
180             end
181             tjc = toc(tsjc);
```

```
182             tsgsc = tic;
183             for i = 1:(length(t)-1)
184                 b = Mcn_p*v_gsc;
185                 [v_gsc, k_gsc, e_gsc] = GS_comp(Mcn,b,
                        v_gsc,K,eps);
186             end
187             tgsc = toc(tsgsc);
188         end
189
190         if (runR)
191             tsrm = tic;
192             for i = 1:(length(t)-1)
193                 b = Mcn_p*v_rm;
194                 [v_rm, k_rm, e_rm] = R_matrix(Mcn,b,v_rm
                        ,alpha,K,eps);
195             end
196             trm = toc(tsrm);
197         end
198         tsjm = tic;
199         for i = 1:(length(t)-1)
200             b = Mcn_p*v_jm;
201             [v_jm, k_jm, e_jm] = J_matrix(Mcn,b,v_jm,K
                    ,eps);
202         end
203         tjm = toc(tsjm);
204         tsgsm = tic;
205         for i = 1:(length(t)-1)
206             b = Mcn_p*v_gsm;
207             [v_gsm, k_gsm, e_gsm] = GS_matrix(Mcn,b,
                    v_gsm,K,eps);
208         end
209         tgsm = toc(tsgsm);
210
211         if runCG
212             tscg = tic;
213             for i = 1:(length(t)-1)
214                 b = Mcn_p*v_cg;
215                 [v_cg, k_cg] = cg(Mcn,b,v_cg,K,eps);
216             end
217             tcg = toc(tscg);
218         end
219
220         tsbicg = tic;
221         for i = 1:(length(t)-1)
222             b = Mcn_p*v_bicg;
```

```
223             [v_bicg, f_bicg,res_bicg, k_bicg] = bicg(Mcn
                    ,b,eps,K,[],[],v_bicg);
224         end
225         tbicg = toc(tsbicg);
226 
227         tsjw = tic;
228         for i = 1:(length(t)-1)
229             b = Mcn_p*v_jw;
230             [v_jw, k_jw, e_jw] = J_wm(Mcn,b,w,v_jw,K,
                    eps);
231         end
232         tjw = toc(tsjw);
233 
234     else
235         if comp
236             if (runR)
237                 tsrc = tic;
238                 for i = 1:(length(t)-1)
239                     [v_rc, k_rc, e_rc] = R_comp(M,v_rc,
                            v_rc,alpha,K,eps);
240                 end
241                 trc = toc(tsrc);
242             end
243             tsjc = tic;
244             for i = 1:(length(t)-1)
245                 [v_jc, k_jc, e_jc] = J_comp(M,v_jc,v_jc
                        ,K,eps);
246             end
247             tjc = toc(tsjc);
248             tsgsc = tic;
249             for i = 1:(length(t)-1)
250                 [v_gsc, k_gsc, e_gsc] = GS_comp(M,v_gsc
                        ,v_gsc,K,eps);
251             end
252             tgsc = toc(tsgsc);
253         end
254 
255         if (runR)
256             tsrm = tic;
257             for i = 1:(length(t)-1)
258             [v_rm, k_rm, e_rm] = R_matrix(M,v_rm,v_rm,
                    alpha,K,eps);
259             end
260             trm = toc(tsrm);
261         end
```

```
262         if Gelim
263             tsie = tic;
264             for i = 1:(length(t)-1)
265                 v_ie = M\v_ie;          % Gauss
                        Elimination
266             end
267             tie = toc(tsie);
268         end
269         tsjm = tic;
270         for i = 1:(length(t)-1)
271             [v_jm, k_jm, e_jm] = J_matrix(M,v_jm,v_jm,
                    K,eps);
272         end
273         tjm = toc(tsjm);
274         tsgsm = tic;
275         for i = 1:(length(t)-1)
276             [v_gsm, k_gsm, e_gsm] = GS_matrix(M,v_gsm,
                    v_gsm,K,eps);
277         end
278         tgsm = toc(tsgsm);
279
280         if runCG
281             tscg = tic;
282             for i = 1:(length(t)-1)
283                 [v_cg, k_cg] = cg(M,v_cg,v_cg,K,eps);
284             end
285             tcg = toc(tscg);
286         end
287
288         tsbicg = tic;
289         for i = 1:(length(t)-1)
290             [v_bicg, f_bicg, res_bicg, k_bicg] = bicg(
                    M,v_bicg,eps,K,[],[],v_bicg);
291         end
292         tbicg = toc(tsbicg);
293
294         tsjw = tic;
295         for i = 1:(length(t)-1)
296             [v_jw, k_jw, e_jw] = J_wm(M,v_jw,w,v_jw,K,
                    eps);
297         end
298         tjw = toc(tsjw);
299     end
300     toc(loop)
301
```

```
302     % analytical solution
303     u = exp((-3*(2*pi)^2/beta)*(t(end)))*v0;
304
305     err_ee_an(q) = norm(v_ee-u(:))/norm(u(:));
306     err_ee_max(q) = max(abs(v_ee-u(:)));
307     time_ee(q) = tee;
308
309     if runIE
310         if Gelim
311             err_ie_an(q) = norm(v_ie-u(:))/norm(u(:));
312             err_ie_max(q) = max(abs(v_ie-u(:)));
313             time_ie(q) = tie;
314         end
315     end
316
317     if (runR)
318         err_rm_an(q) = norm(v_rm-u(:))/norm(u(:));
319         err_rm_max(q) = max(abs(v_rm-u(:)));
320         krm(q) = k_rm;
321         time_rm(q) = trm;
322     end
323
324     err_jm_an(q) = norm(v_jm-u(:))/norm(u(:));
325     err_jm_max(q) = max(abs(v_jm-u(:)));
326     kjm(q) = k_jm;
327     time_jm(q) = tjm;
328     err_gsm_an(q) = norm(v_gsm-u(:))/norm(u(:));
329     err_gsm_max(q) = max(abs(v_gsm-u(:)));
330     kgsm(q) = k_gsm;
331     time_gsm(q) = tgsm;
332
333     if runCG
334         err_cg_an(q) = norm(v_cg-u(:))/norm(u(:));
335         err_cg_max(q) = max(abs(v_cg-u(:)));
336         kcg(q) = k_cg;
337         time_cg(q) = tcg;
338     end
339
340     err_bicg_an(q) = norm(v_bicg-u(:))/norm(u(:));
341     err_bicg_max(q) = max(abs(v_bicg-u(:)));
342     kbicg(q) = k_bicg;
343     time_bicg(q) = tbicg;
344
345     err_jw_an(q) = norm(v_jw-u(:))/norm(u(:));
346     err_jw_max(q) = max(abs(v_jw-u(:)));
```

```
347         kjw(q) = k_jw;
348         time_jw(q) = tjw;
349 
350         if comp
351             if (runR)
352                 krc(q) = k_rc;
353                 time_rc(q) = trc;
354             end
355             kjc(q) = k_jc;
356             time_jc(q) = tjc;
357             kgsc(q) = k_gsc;
358             time_gsc(q) = tgsc;
359         end
360 
361         pause(0.01)
362 
363     end
364     toc
365 
366     %Code for printing and plotting results removed
367     end
```

Appendix C

Parallel Implementaion in C++

cpp/Jacobi.h

```
1  #include <iostream>
2  #include <string>
3
4  #include "Epetra_Map.h"
5  #include "Epetra_CrsMatrix.h"
6  #include "Epetra_Vector.h"
7  #include "AztecOO.h"
8
9  #include "ml_include.h"
10 #include "Epetra_LinearProblem.h"
11 #include "ml_MultiLevelOperator.h"
12 #include "ml_epetra_utils.h"
13
14 #ifdef HAVE_MPI
15 #include "mpi.h"
16 #include "Epetra_MpiComm.h"
17 #else
18 #include "Epetra_SerialComm.h"
19 #endif
20
21 class Jacobi
22 {
23     private:
24         int *subd_rank, *subd_lo_ix, *subd_hi_ix, *
               global_num_cells, *num_parts;
25         int p_up, p_down, p_over, p_under, p_left,
               p_right;
26         int *gi_left, *gi_right, *gi_down, *gi_up, *
               gi_under, *gi_over;
27         int *gi_send_left, *gi_send_right, *
               gi_send_down, *gi_send_up, *gi_send_under,
```

```
            *gi_send_over;
28          double *send_left, *send_right, *send_down, *
               send_up, *send_under, *send_over;
29          double *recv_left, *recv_right, *recv_down, *
               recv_up, *recv_under, *recv_over;
30          int i_min,i_max,j_min,j_max,k_min,k_max;
31          int x_start, actual_nx, y_start, actual_ny,
               z_start, actual_nz;
32
33      protected:
34          double *u, *u_prev;
35          int *global_indices, *ext_global_indices,
               num_local_parts, num_ext_parts;
36          int nsd, num_procs, my_id, max_it, test_conv;
37          double dt, delta, theta, dx, dy, dz, tol;
38          Epetra_Map *ext_map, *map;
39  #ifdef HAVE_MPI
40          Epetra_MpiComm *comm;
41          Epetra_MpiComm *comm2;
42  #else
43          Epetra_SerialComm *comm;
44          Epetra_SerialComm *comm2;
45  #endif
46
47      public:
48          double *u_end;
49
50          Jacobi();
51          ~Jacobi();
52
53          void SetParameters(int nsd, int*
               global_num_cells, int* num_parts,
54                  int* subd_lo_ix, int* subd_hi_ix, int*
                       subd_rank, int* global_indices,
55                  int num_local_parts, double dt, double
                       delta);
56
57          void SolveJacobi(Epetra_CrsMatrix* A,
               Epetra_Vector* b, Epetra_Vector* u0,
58                  int max_it, double tol, int test_conv)
                       ;
59
60  };
```

cpp/Jacobi.cpp

```
#include <malloc.h>
#include <Jacobi.h>

Jacobi:: Jacobi ()
{
    //This class is hardcoded to only solve 3D
       problems
#ifdef HAVE_MPI
    MPI_Comm_rank (MPI_COMM_WORLD, &my_id);
    MPI_Comm_size (MPI_COMM_WORLD, &num_procs);
#else
    my_id = 0;
#endif
}

Jacobi:: ~Jacobi ()
{
  //Deconstructor Code hidden
}

void Jacobi:: SetParameters(int nsd_, int*
   global_num_cells_, int* num_parts_,
        int* subd_lo_ix_, int* subd_hi_ix_, int*
           subd_rank_, int* global_indices_,
        int num_local_parts_, double dt_, double
           delta_) {

  //Set general parameters
    int i,j,k,l;
    nsd = nsd_;
    dt = dt_;
    delta = delta_;
    num_local_parts = num_local_parts_;
    global_num_cells = (int*) malloc(nsd*sizeof(int));
    subd_rank = (int*) malloc(nsd*sizeof(int));
    subd_lo_ix = (int*) malloc(nsd*sizeof(int));
    subd_hi_ix = (int*) malloc(nsd*sizeof(int));
    num_parts = (int*) malloc(nsd*sizeof(int));

    for (i=0; i<nsd; i++) {
        global_num_cells[i] = global_num_cells_[i];
        subd_lo_ix[i] = subd_lo_ix_[i];
        subd_hi_ix[i] = subd_hi_ix_[i];
        num_parts[i] = num_parts_[i];
        subd_rank[i] = subd_rank_[i];
```

```
42      }
43
44    //Set the ID of neighbouring processes in each
          direction
45      p_left = subd_rank[0] != 0 ? my_id-1 :
          MPI_PROC_NULL;
46      p_right = subd_rank[0] != (num_parts[0]-1) ? my_id
          +1 : MPI_PROC_NULL;
47      p_down = subd_rank[1] != 0 ? my_id-num_parts[0] :
          MPI_PROC_NULL;
48      p_up = subd_rank[1] != (num_parts[1]-1) ? my_id+
          num_parts[0] : MPI_PROC_NULL;
49      p_under = subd_rank[2] != 0 ? my_id-(num_parts[0]*
          num_parts[1]) : MPI_PROC_NULL;
50      p_over = subd_rank[2] != (num_parts[2]-1) ? my_id
          +(num_parts[0]*num_parts[1]) : MPI_PROC_NULL;
51      global_indices = (int*)malloc(num_local_parts*
          sizeof(int));
52      for (i=0; i<num_local_parts; i++) {
53          global_indices[i] = global_indices_[i];
54      }
55
56    //Set expanded domain in each direction
57      i_min = p_left != MPI_PROC_NULL ? subd_lo_ix[0]-1
          : subd_lo_ix[0];
58      i_max = p_right != MPI_PROC_NULL ? subd_hi_ix[0]+1
          : subd_hi_ix[0];
59      j_min = p_down != MPI_PROC_NULL ? subd_lo_ix[1]-1
          : subd_lo_ix[1];
60      j_max = p_up != MPI_PROC_NULL ? subd_hi_ix[1]+1 :
          subd_hi_ix[1];
61      k_min = p_under != MPI_PROC_NULL ? subd_lo_ix[2]-1
          : subd_lo_ix[2];
62      k_max = p_over != MPI_PROC_NULL ? subd_hi_ix[2]+1
          : subd_hi_ix[2];
63      num_ext_parts = (i_max-i_min+1)*(j_max-j_min+1)*(
          k_max-k_min+1);
64
65      if (nsd == 2) {
66          num_ext_parts = (i_max-i_min+1)*(j_max-j_min
              +1);
67      }
68
69      ext_global_indices = (int*)malloc(num_ext_parts*
          sizeof(int));
```

```
70      const int Nx = global_num_cells[0]+1;
71      const int Ny = global_num_cells[1]+1;
72      const int Nz = global_num_cells[2]+1;
73
74      const int nx = subd_hi_ix[0]-subd_lo_ix[0]+1;
75      const int ny = subd_hi_ix[1]-subd_lo_ix[1]+1;
76      const int nz = subd_hi_ix[2]-subd_lo_ix[2]+1;
77      const int y_offset = global_num_cells[0]+1;
78      const int z_offset = (global_num_cells[0]+1)*(
           global_num_cells[1]+1);
79
80
81      l = 0;
82      for (k=k_min; k<=k_max; k++) {
83          for (j=j_min; j<=j_max; j++) {
84              for (i=i_min; i<=i_max; i++) {
85                  ext_global_indices[l++] = k*Nx*Ny+j*Nx
                       +i; // 3D logic
86              }
87          }
88      }
89
90    //Vectors needed for indices and temporary storage
       when communicating with neighbouring processes
91      gi_left = new int[ny*nz];
92      gi_right = new int[ny*nz];
93      gi_down = new int[nx*nz];
94      gi_up = new int[nx*nz];
95      gi_over = new int[nx*ny];
96      gi_under = new int[nx*ny];
97      gi_send_left = new int[ny*nz];
98      gi_send_right = new int[ny*nz];
99      gi_send_down = new int[nx*nz];
100     gi_send_up = new int[nx*nz];
101     gi_send_over = new int[nx*ny];
102     gi_send_under = new int[nx*ny];
103
104     send_left = new double[ny*nz];
105     send_right = new double[ny*nz];
106     send_up = new double[nx*nz];
107     send_down = new double[nx*nz];
108     send_over = new double[nx*ny];
109     send_under = new double[nx*ny];
110
111     recv_left = new double[ny*nz];
```

```
112     recv_right = new double[ny*nz];
113     recv_up = new double[nx*nz];
114     recv_down = new double[nx*nz];
115     recv_over = new double[nx*ny];
116     recv_under = new double[nx*ny];
117 
118     x_start = 0, actual_nx = nx;
119     y_start = 0, actual_ny = ny;
120     z_start = 0, actual_nz = nz;
121 
122   //Set if ghost rows needed in each direction
123     if (p_left != MPI_PROC_NULL) {
124         x_start = 1;
125         actual_nx += 1;
126     }
127     if (p_right != MPI_PROC_NULL) {
128         actual_nx += 1;
129     }
130     if (p_down != MPI_PROC_NULL) {
131         y_start = 1;
132         actual_ny += 1;
133     }
134     if (p_up != MPI_PROC_NULL) {
135         actual_ny += 1;
136     }
137     if (p_under != MPI_PROC_NULL) {
138         z_start = 1;
139         actual_nz += 1;
140     }
141     if (p_over != MPI_PROC_NULL) {
142         actual_nz += 1;
143     }
144 
145     u = (double*)malloc(actual_nx*actual_ny*actual_nz*
        sizeof(double));
146     u_prev = (double*)malloc(actual_nx*actual_ny*
        actual_nz*sizeof(double));
147     u_end = (double*)malloc(nx*ny*nz*sizeof(double));
148 
149   //Calculate coordinates for data to be sent and
        received from processes i-1 and i+1
150     int ic = 0;
151     for (k = z_start; k < z_start+nz; k++) {
152         for (j = y_start; j < y_start+ny; j++) {
153             gi_send_left[ic] = k*(actual_nx*actual_ny)
```

```
                    + j*actual_nx + 1;
154                 gi_send_right[ic] = k*(actual_nx*actual_ny
                        ) + j*actual_nx + (actual_nx-2);
155                 gi_left[ic] = k*(actual_nx*actual_ny) + j*
                        actual_nx + 0;
156                 gi_right[ic] = k*(actual_nx*actual_ny) + j
                        *actual_nx + (actual_nx-1);
157                 ic++;
158             }
159         }
160     //Calculate coordinates for data to be sent and
            received from processes j-1 and j+1
161         ic = 0;
162         for (k = z_start; k < z_start+nz; k++) {
163             for (i = x_start; i<x_start+nx; i++) {
164                 gi_send_down[ic] = k*(actual_nx*actual_ny)
                        + 1*actual_nx + i;
165                 gi_send_up[ic] = k*(actual_nx*actual_ny) +
                        (actual_ny-2)*actual_nx + i;
166                 gi_down[ic] = k*(actual_nx*actual_ny) + 0*
                        actual_nx + i;
167                 gi_up[ic] = k*(actual_nx*actual_ny) + (
                        actual_ny-1)*actual_nx + i;
168                ic++;
169             }
170         }
171     //Calculate coordinates for data to be sent and
            received from processes k-1 and k+1
172         ic = 0;
173         for (j = y_start; j<y_start+ny; j++) {
174             for (i = x_start; i<x_start+nx; i++) {
175                 gi_send_under[ic] = 1*(actual_nx*actual_ny
                        ) + j*actual_nx + i;
176                 gi_send_over[ic] = (actual_nz-2)*(
                        actual_nx*actual_ny) + j*actual_nx + i;
177                 gi_under[ic] = 0*(actual_nx*actual_ny) + j
                        *actual_nx + i;
178                 gi_over[ic] = (actual_nz-1)*(actual_nx*
                        actual_ny) + j*actual_nx + i;
179                ic++;
180
181             }
182         }
183 }
184
```

```
185
186 void Jacobi:: SolveJacobi(Epetra_CrsMatrix* A,
      Epetra_Vector* b, Epetra_Vector* u0,
187           int max_it_, double tol_, int test_conv_) {
188
189
190     int i,j,k,l,n, ic, gi, li;
191     max_it = max_it_;
192     tol = tol_;
193     test_conv = test_conv_;
194     const int Nx = global_num_cells[0]+1;
195     const int Ny = global_num_cells[1]+1;
196     const int Nz = global_num_cells[2]+1;
197
198     const int nx = subd_hi_ix[0]-subd_lo_ix[0]+1;
199     const int ny = subd_hi_ix[1]-subd_lo_ix[1]+1;
200     const int nz = subd_hi_ix[2]-subd_lo_ix[2]+1;
201     const int y_offset = global_num_cells[0]+1;
202     const int z_offset = (global_num_cells[0]+1)*(
          global_num_cells[1]+1);
203
204     int max_coord = Nx*Ny*Nz;
205     double global_diff, diff, tdiff;
206
207   //Storing the initial guess vector in u_prev
208   li = 0;
209     for (k = z_start; k<z_start+nz; k++) {
210         for (j = y_start; j<y_start+ny; j++) {
211             for (i = x_start; i<x_start+nx; i++) {
212                 ic = k*(actual_nx*actual_ny) + j*
                      actual_nx + i;
213                 u_prev[ic] = (*u0)[li];
214                 li++;
215             }
216         }
217     }
218
219     l = 0;
220     diff = 0.0;
221     int gj, lc;
222     double s, a, upr;
223     int GlobalRow,MyRow,NumNzRow,NumEntries;
224
225   //The Jacobi Iteration
226     for (n=0; n<=max_it; n++) {
```

```
227
228         //Extracting correct values to send to each
                neighbour
229             if (p_left != MPI_PROC_NULL) {
230                 for (k = 0; k<ny*nz; k++) {
231                     send_left[k] = u_prev[gi_send_left[k
                            ]];
232                 }
233             }
234             if (p_right != MPI_PROC_NULL) {
235                 for (k = 0; k<ny*nz; k++) {
236                     send_right[k] = u_prev[gi_send_right[k
                            ]];
237                 }
238             }
239             if (p_up != MPI_PROC_NULL) {
240                 for (k = 0; k<nx*nz; k++) {
241                     send_up[k] = u_prev[gi_send_up[k]];
242                 }
243             }
244             if (p_down != MPI_PROC_NULL)  {
245                 for (k = 0; k<nx*nz; k++) {
246                     send_down[k] = u_prev[gi_send_down[k
                            ]];
247                 }
248             }
249             if (p_over != MPI_PROC_NULL) {
250                 for (k = 0; k<nx*ny; k++) {
251                     send_over[k] = u_prev[gi_send_over[k
                            ]];
252                 }
253             }
254             if (p_under != MPI_PROC_NULL) {
255                 for (k = 0; k<nx*ny; k++) {
256                     send_under[k] = u_prev[gi_send_under[k
                            ]];
257                 }
258             }
259         //Sending and receiving needed values of u_prev
260             MPI_Barrier(MPI_COMM_WORLD);
261             MPI_Sendrecv(send_left,ny*nz,MPI_DOUBLE,p_left
                    ,my_id,
262                     recv_right,ny*nz,MPI_DOUBLE,p_right,
                            p_right,MPI_COMM_WORLD,
                            MPI_STATUS_IGNORE);
```

```
263         MPI_Sendrecv(send_right,ny*nz,MPI_DOUBLE,
                p_right,my_id,
264                 recv_left,ny*nz,MPI_DOUBLE,p_left,
                        p_left,MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
265         MPI_Sendrecv(send_up,nx*nz,MPI_DOUBLE,p_up,
                my_id,
266                 recv_down,nx*nz,MPI_DOUBLE,p_down,
                        p_down,MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
267         MPI_Sendrecv(send_down,nx*nz,MPI_DOUBLE,p_down
                ,my_id,
268                 recv_up,nx*nz,MPI_DOUBLE,p_up,p_up,
                        MPI_COMM_WORLD,MPI_STATUS_IGNORE);
269         MPI_Sendrecv(send_over,nx*ny,MPI_DOUBLE,p_over
                ,my_id,
270                 recv_under,nx*ny,MPI_DOUBLE,p_under,
                        p_under,MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
271         MPI_Sendrecv(send_under,nx*ny,MPI_DOUBLE,
                p_under,my_id,
272                 recv_over,nx*ny,MPI_DOUBLE,p_over,
                        p_over,MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
273         MPI_Barrier(MPI_COMM_WORLD);
274
275     //Storing the received values in the correct ghost
            cells
276         if (p_left != MPI_PROC_NULL) {
277             for (k = 0; k<ny*nz; k++) {
278                 u_prev[gi_left[k]] = recv_left[k];
279             }
280         }
281         if (p_right != MPI_PROC_NULL) {
282             for (k = 0; k<ny*nz; k++) {
283                 u_prev[gi_right[k]] = recv_right[k];
284             }
285         }
286         if (p_up != MPI_PROC_NULL) {
287             for (k = 0; k<nx*nz; k++) {
288                 u_prev[gi_up[k]] = recv_up[k];
289             }
290         }
291         if (p_down != MPI_PROC_NULL) {
292             for (k = 0; k<nx*nz; k++) {
```

```
293                     u_prev[gi_down[k]] = recv_down[k];
294                 }
295             }
296             if (p_over != MPI_PROC_NULL) {
297                 for (k = 0; k<nx*ny; k++) {
298                     u_prev[gi_over[k]] = recv_over[k];
299                 }
300             }
301             if (p_under != MPI_PROC_NULL) {
302                 for (k = 0; k<nx*ny;k++) {
303                     u_prev[gi_under[k]] = recv_under[k];
304                 }
305             }
306 
307             MPI_Barrier(MPI_COMM_WORLD);
308             tdiff = 0.0;
309             MyRow = 0;
310             diff = 0.0;
311         //Solving each equation using Jacobi
312             for (k = z_start; k<z_start+nz; k++) {
313                 for (j = y_start; j<y_start+ny; j++) {
314                     for (i = x_start; i<x_start+nx; i++) {
315                         ic = k*(actual_nx*actual_ny) + j*
                               actual_nx + i; //Local index
316 
317                         GlobalRow = A->GRID(MyRow);
318 
319                         NumNzRow = A->NumMyEntries(MyRow);
320                         double *Values = new double[
                               NumNzRow];
321                         int *LocalIndices = new int[
                               NumNzRow];
322 
323                         A->ExtractMyRowCopy(MyRow,NumNzRow
                               ,NumEntries,Values,LocalIndices
                               );
324 
325                         s = 0;
326                         a = 1.0;
327                         gi = MyRow;
328 
329                   //Calculating the correct sum for this row
330                         for (lc=0; lc<NumEntries; ++lc) {
331                             gj = A->GCID(LocalIndices[lc])
                                   ;
```

```
332                     if (gj == GlobalRow - z_offset
                            ) {
333                       s += Values[lc]*u_prev[ic
                              -(actual_nx*actual_ny)
                              ];
334                     } else if (gj == GlobalRow -
                            y_offset) {
335                       s += Values[lc]*u_prev[ic-
                              actual_nx];
336                     } else if (gj == GlobalRow -
                            1) {
337                       s += Values[lc]*u_prev[ic
                              -1];
338                     } else if (gj == GlobalRow) {
339                       a = 1.0/Values[lc];
340                     } else if (gj == GlobalRow +
                            1) {
341                       s += Values[lc]*u_prev[ic
                              +1];
342                     } else if (gj == GlobalRow +
                            y_offset) {
343                       s += Values[lc]*u_prev[ic+
                              actual_nx];
344                     } else if (gj = GlobalRow +
                            z_offset) {
345                       s += Values[lc]*u_prev[ic
                              +(actual_nx*actual_ny)
                              ];
346                     } else {
347                       printf("<%03d> Something
                              went wrong with the
                              indices\n", my_id); ;
348                     }
349                   }
350                   u[ic] = ((*b)[gi] - s)*a;
351                   if (fabs(u[ic]-u_prev[ic]) > tdiff
                          ) {
352                     tdiff = fabs(u[ic]-u_prev[ic])
                            ;
353                   }
354
355                   delete Values;
356                   delete LocalIndices;
357                   MyRow++;
358                 }
```

```
359                 }
360             }
361
362             diff = tdiff;
363
364         MPI_Barrier(MPI_COMM_WORLD);
365         if ((n%test_conv) == 0) {
366           //test if convergence is reached. If true,
                return
367           if (num_procs == 1) {
368             global_diff = diff;
369           } else {
370             MPI_Allreduce(&diff, &global_diff, 1,
                  MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
371           }
372           if (global_diff <= tol) {
373             printf("<%03d> Convergence reached. n=%d, Diff
                  =%g. Tol=%g. Returning\n", my_id, n,
                  global_diff, tol);
374             gi = 0;
375             for (k=z_start; k<z_start+nz; k++) {
376               for (j=y_start; j<y_start+ny; j++) {
377                 for (i=x_start; i<x_start+nx; i++) {
378                   ic = k*(actual_nx*actual_ny) + j*
                        actual_nx + i;
379                   u_end[gi] = u[ic];
380                   gi++;
381                 }
382               }
383             }
384             return;
385           }
386         }
387
388         for (j=0; j<num_ext_parts; j++) {
389           u_prev[j] = u[j];
390         }
391       }
392       printf("<%03d> Conv not reached within iteration
            limit N=%d reached. Diff=%g. Tol=%g\n", my_id,
            max_it,global_diff, tol);
393       gi = 0;
394       for (k=z_start; k<z_start+nz; k++) {
395         for (j=y_start; j<y_start+ny; j++) {
396           for (i=x_start; i<x_start+nx; i++) {
```

```
397             ic = k*(actual_nx*actual_ny) + j*actual_nx + i
                    ;
398             u_end[gi] = u[ic];
399             gi++;
400           }
401         }
402       }
403     }
```

cpp/RectDomain.h

```
1  #include <iostream>
2  #include <string>
3
4  #ifndef Rectangular_Domain_h__
5  #define Rectangular_Domain_h__
6
7  #include "Epetra_Map.h"
8  #include "Epetra_CrsMatrix.h"
9  #include "Epetra_Vector.h"
10 #include "AztecOO.h"
11
12 #include "ml_include.h"
13 #include "Epetra_LinearProblem.h"
14 #include "ml_MultiLevelOperator.h"
15 #include "ml_epetra_utils.h"
16
17 #ifdef HAVE_MPI
18 #include "mpi.h"
19 #include "Epetra_MpiComm.h"
20 #else
21 #include "Epetra_SerialComm.h"
22 #endif
23
24
25 class RectDomain
26 {
27 private:
28   int *subd_rank, *subd_lo_ix, *subd_hi_ix, *
       global_num_cells;
29
30 protected:
31   Epetra_Vector *u, *u_prev, *rhs;
32   Epetra_CrsMatrix *A;
33   Epetra_LinearProblem *linear_system;
34   AztecOO *linear_solver;
```

```
35
36    ML *ml_handle;
37    ML_Aggregate *agg_object;
38
39    Epetra_Map *map;
40 #ifdef HAVE_MPI
41    Epetra_MpiComm *comm;
42 #else
43    Epetra_SerialComm *comm;
44 #endif
45    ML_Epetra::MultiLevelOperator *MLop;
46
47    int *global_indices, num_local_pts;
48
49    int nsd, num_procs, my_id;
50    double *scaling_vec, dt, delta, theta, dx, dy, dz;
51
52    void saveResults (double* vec, const char* v_name,
         double time);
53
54  public:
55    RectDomain ();
56   ~RectDomain ();
57
58    void preparePartition (int nsd, int*
         global_num_cells, int* num_parts);
59    void prepareMatrix (double dt, double delta, double
         theta, double* spacing, int nsd_, char*
         solver_type);
60    void preparePreconditioner (char* solver_type);
61
62    void work (double dt_, double T_,  int*
         global_num_cells, int* num_parts, char*
         solver_type, int nsd_);
63  };
64
65  #endif
```

cpp/RectDomain.cpp

```
1  #include <RectDomain.h>
2  #include <Panfilov_2order.h>
3  #include <Jacobi.cpp>
4  #include <malloc.h>
5
6  RectDomain::RectDomain ()
```

```
7  {
8  #ifdef HAVE_MPI
9      MPI_Comm_rank (MPI_COMM_WORLD, &my_id);
10 #else
11     my_id = 0;
12 #endif
13 }
14
15 RectDomain:: ~RectDomain ()
16 {
17     free (global_indices);
18     free (subd_rank);
19     free (subd_lo_ix);
20     free (subd_hi_ix);
21     free (global_num_cells);
22     free (scaling_vec);
23 }
24
25 void RectDomain:: preparePartition (int nsd_,
26         int* global_num_cells_,
27         int* num_parts)
28 {
29     int num_subds = 1, i, j, ix, k, m, l, rank;
30
31     //nsd = nsd_;
32     int* offsets = (int*) malloc(nsd*sizeof(int));
33
34     global_num_cells = (int*) malloc(nsd*sizeof(int));
35     subd_rank = (int*) malloc(nsd*sizeof(int));
36     subd_lo_ix = (int*) malloc(nsd*sizeof(int));
37     subd_hi_ix = (int*) malloc(nsd*sizeof(int));
38
39     for (i=0; i<nsd; i++) {
40         global_num_cells[i] = global_num_cells_[i];
41         num_subds *= num_parts[i];
42     }
43
44     offsets[0] = 1;
45     subd_rank[0] = my_id%num_parts[0];
46
47     if (nsd>=2) {
48         offsets[1] = num_parts[0];
49         subd_rank[1] = my_id/offsets[1];
50     }
51
```

```
52      if (nsd==3) {
53          offsets[2] = num_parts[0]*num_parts[1];
54          subd_rank[1] = (my_id%offsets[2])/num_parts
                [0];
55          subd_rank[2] = my_id/offsets[2];
56      }
57
58      num_local_pts = 1;
59      printf("my_id=%d, gnum_cells=[%d,%d,%d] parts=[%d
            ,%d,%d]\n",
60              my_id, global_num_cells[0],
                    global_num_cells[1],global_num_cells
                    [2],num_parts[0],num_parts[1],num_parts
                    [2]);
61
62      for (i=0; i<nsd; i++) {
63          rank = subd_rank[i];
64
65          int num_pts = global_num_cells[i]+1;
66
67          k = num_pts/num_parts[i];
68          m = num_pts%num_parts[i];
69
70          ix = rank*k + ((m<rank) ? m : rank);
71          subd_lo_ix[i] = ix;
72
73          ix = ix+k-1;
74          if (rank<m)
75              ix = ix+1;
76          subd_hi_ix[i] = ix;
77          printf("my_id=%d, i=%d, rank=%d, lo_ix=%d,
                hi_ix=%d\n",
78                  my_id,i,rank,subd_lo_ix[i],ix);
79
80          num_local_pts *= (subd_hi_ix[i]-subd_lo_ix[i
                ]+1);
81      }
82
83  #ifdef HAVE_MPI
84      MPI_Barrier (MPI_COMM_WORLD);
85  #endif
86
87      global_indices = (int*)malloc(num_local_pts*sizeof
            (int));
88      k = 0;
```

```
89      if (nsd==3) {
90          for (l=subd_lo_ix[2]; l<=subd_hi_ix[2]; l++) {
91              for (j=subd_lo_ix[1]; j<=subd_hi_ix[1]; j
                    ++) {
92                  for (i=subd_lo_ix[0]; i<=subd_hi_ix
                        [0]; i++) {
93                      global_indices[k++] = l*(
                            global_num_cells[0]+1)*(
                            global_num_cells[1]+1)
94                           +j*(global_num_cells[0]+1)+i;
                                // 3D logic
95                  }
96              }
97          }
98      } else if (nsd == 2) {
99          for (j=subd_lo_ix[1]; j<=subd_hi_ix[1]; j++)
100             for (i=subd_lo_ix[0]; i<=subd_hi_ix[0]; i
                    ++)
101                 global_indices[k++] = j*(
                        global_num_cells[0]+1)+i;
102     }
103
104 #ifdef HAVE_MPI
105     comm = new Epetra_MpiComm ( MPI_COMM_WORLD );
106 #else
107     comm = new Epetra_SerialComm;
108 #endif
109     map = new Epetra_Map (-1,num_local_pts,
            global_indices,0,*comm);
110 }
111
112 void RectDomain::prepareMatrix (double dt_, double
        delta_, double theta_,
113         double* spacing, int nds_, char* solver_type)
114 {
115     dt = dt_;
116     delta = delta_;
117     theta = theta_;
118     //nsd = nds_;
119
120     dx = spacing[0]; dy = spacing[1]; dz = 1;
121     if (nsd >= 3)
122         dz = spacing[2];
123
124     if (nsd == 2)
```

```
125         A = new Epetra_CrsMatrix (Copy, *map, 5);   //
                2D finite difference
126     else if (nsd == 3)
127         A = new Epetra_CrsMatrix (Copy, *map, 7);   //
                3D finite difference
128
129     const int offset = global_num_cells[0]+1;
130     const double z_offset = (global_num_cells[0]+1)*(
            global_num_cells[1]+1);
131
132     const double vx = -dt*theta*delta/dx/dx;
133     const double vy = -dt*theta*delta/dy/dy;
134     const double vz = -dt*theta*delta/dz/dz;
135
136     double value_center;
137     if (nsd == 2) {
138         value_center = 1.0+2.*dt*theta*delta/dx/dx+2.*
                dt*theta*delta/dy/dy;
139     } else if (nsd == 3) {
140         value_center = 1.0 + 2.*dt*theta*delta/dx/dx +
                2.*dt*theta*delta/dy/dy
141             + 2.*dt*theta*delta/dz/dz;
142     }
143     double v, value_x, value_y, value_z;
144
145     int i,j,k,l, gi,pos;
146     scaling_vec = (double*)malloc(num_local_pts*sizeof
            (double));
147     k = 0;
148     if (nsd==3) {
149         for (l=subd_lo_ix[2]; l<=subd_hi_ix[2]; l++) {
150             for (j=subd_lo_ix[1]; j<=subd_hi_ix[1]; j
                    ++) {
151                 for (i=subd_lo_ix[0]; i<=subd_hi_ix
                        [0]; i++) {
152                     scaling_vec[k] = 1.0;
153                     if (i==0 || i==
                            global_num_cells[0])
154                         scaling_vec[k] *= 0.5;
155                     if (j==0 || j==
                            global_num_cells[1])
156                         scaling_vec[k] *= 0.5;
157                     if (l==0 || l==
                            global_num_cells[2])
158                         scaling_vec[k] *= 0.5;
```

```
159                         ++k;
160                     }
161                 }
162             }
163         } else if (nsd==2) {
164             for (j=subd_lo_ix[1]; j<=subd_hi_ix[1]; j++) {
165                 for (i=subd_lo_ix[0]; i<=subd_hi_ix[0]; i
                        ++) {
166                     scaling_vec[k] = 1.0;
167                     if (strcmp(solver_type, "CG")==0) {
168                         if (i==0 || i==global_num_cells
                                [0])
169                             scaling_vec[k] *= 0.5;
170                         if (j==0 || j==global_num_cells
                                [1])
171                             scaling_vec[k] *= 0.5;
172                     }
173                     ++k;
174                 }
175             }
176         }
177 
178         k = 0;
179         if (nsd==3) {
180             for (l=subd_lo_ix[2]; l<=subd_hi_ix[2]; l++) {
181 
182                 if (l==0 || l==global_num_cells[2])
183                     value_z = 2.0*vz;
184                 else
185                     value_z = vz;
186 
187                 for (j=subd_lo_ix[1]; j<=subd_hi_ix[1]; j
                        ++) {
188 
189                     if (j==0 || j==global_num_cells[1])
190                         value_y = 2.0*vy;
191                     else
192                         value_y = vy;
193 
194                     for (i=subd_lo_ix[0]; i<=subd_hi_ix
                            [0]; i++) {
195 
196                         if (i==0 || i==global_num_cells
                                [0])
197                             value_x = 2.0*vx;
```

```
198                 else
199                     value_x = vx;
200
201                 gi = global_indices[k];
202
203                 if (l!=0) {
204                     v = scaling_vec[k]*value_z;
205                     pos = gi-z_offset;
206                     A->InsertGlobalValues(gi,1,&v,
                            &pos);
207                 }
208                 if (j!=0) {
209                     v = scaling_vec[k]*value_y;
210                     pos = gi-offset;
211                     A->InsertGlobalValues(gi, 1, &
                            v, &pos);
212                 }
213                 if (i!=0) {
214                     v = scaling_vec[k]*value_x;
215                     pos = gi-1;
216                     A->InsertGlobalValues(gi, 1, &
                            v, &pos);
217                 }
218                 v = scaling_vec[k]*value_center;
219                 A->InsertGlobalValues(gi, 1, &v, &
                    gi);
220                 if (i!=global_num_cells[0]) {
221                     v = scaling_vec[k]*value_x;
222                     pos = gi+1;
223                     A->InsertGlobalValues(gi, 1, &
                            v, &pos);
224                 }
225                 if (j!=global_num_cells[1]) {
226                     v = scaling_vec[k]*value_y;
227                     pos = gi+offset;
228                     A->InsertGlobalValues(gi, 1, &
                            v, &pos);
229                 }
230                 if (l!=global_num_cells[2]) {
231                     v = scaling_vec[k]*value_z;
232                     pos = gi + z_offset;
233                     A->InsertGlobalValues(gi, 1, &
                            v, &pos);
234                 }
235
```

```
236                     ++k;
237                 }
238             }
239         }
240     } else if (nsd == 2) {
241         for (j=subd_lo_ix[1]; j<=subd_hi_ix[1]; j++) {
242 
243             if (j==0 || j==global_num_cells[1])
244                 value_y = 2.0*vy;
245             else
246                 value_y = vy;
247 
248             for (i=subd_lo_ix[0]; i<=subd_hi_ix[0]; i
                ++) {
249 
250                 if (i==0 || i==global_num_cells[0])
251                     value_x = 2.0*vx;
252                 else
253                     value_x = vx;
254 
255                 gi = global_indices[k];
256 
257                 if (j!=0) {
258                     v = scaling_vec[k]*value_y;
259                     pos = gi-offset;
260                     A->InsertGlobalValues(gi, 1, &v, &
                        pos);
261                 }
262                 if (i!=0) {
263                     v = scaling_vec[k]*value_x;
264                     pos = gi-1;
265                     A->InsertGlobalValues(gi, 1, &v, &
                        pos);
266                 }
267                 v = scaling_vec[k]*value_center;
268                 A->InsertGlobalValues(gi, 1, &v, &gi);
269                 if (i!=global_num_cells[0]) {
270                     v = scaling_vec[k]*value_x;
271                     pos = gi+1;
272                     A->InsertGlobalValues(gi, 1, &v, &
                        pos);
273                 }
274                 if (j!=global_num_cells[1]) {
275                     v = scaling_vec[k]*value_y;
276                     pos = gi+offset;
```

```
277                         A->InsertGlobalValues(gi, 1, &v, &
                                pos);
278                     }
279
280                     ++k;
281                 }
282             }
283         }
284
285
286         A->FillComplete();
287
288         u = new Epetra_Vector (*map);
289         u_prev = new Epetra_Vector (*map);
290         rhs = new Epetra_Vector (*map);
291
292 }
293
294 void RectDomain:: preparePreconditioner (char*
        solver_type)
295 {
296         int N_levels = 10;
297         ML_Set_PrintLevel(3);
298         ML_Create(&ml_handle,N_levels);
299
300         // wrap Epetra Matrix into ML matrix (data is NOT
                copied)
301         EpetraMatrix2MLMatrix(ml_handle, 0, A);
302
303         // as we are interested in smoothed aggregation,
                create a ML_Aggregate object
304         // to store the aggregates
305         ML_Aggregate_Create(&agg_object);
306
307         // specify max coarse size
308         ML_Aggregate_Set_MaxCoarseSize(agg_object,1);
309
310         // generate the hierady
311         N_levels = ML_Gen_MGHierarchy_UsingAggregation(
                ml_handle, 0,
312                     ML_INCREASING, agg_object);
313
314         // Set a symmetric Gauss-Seidel smoother for the
                MG method (change
315         // if the matrix is not symmetric)
```

```
316         ML_Gen_Smoother_SymGaussSeidel(ml_handle,
                ML_ALL_LEVELS,
317                 ML_BOTH, 1, ML_DEFAULT);
318
319
320         // generate solver
321         ML_Gen_Solver (ml_handle, ML_MGV, 0, N_levels-1);
322
323         // wrap ML_Operator into Epetra_Operator
324         MLop = new ML_Epetra::MultiLevelOperator (
                ml_handle,*comm,*map,*map);
325
326         // linear system and solver part
327         linear_system = new Epetra_LinearProblem (A, u,
                rhs);
328
329         linear_solver = new AztecOO (*linear_system);
330         if (strcmp(solver_type,"cg")==0 || strcmp(
                solver_type, "CG")==0) {
331           linear_solver->SetAztecOption(AZ_solver, AZ_cg
                  );
332         } else {
333           linear_solver->SetAztecOption(AZ_solver,
                  AZ_gmres);
334         }
335         linear_solver->SetPrecOperator(MLop);
336 }
337
338 void RectDomain::work (double dt_, double T_, int*
        global_num_cells, int* num_parts, char* solver_type
        , int nsd_)
339 {
340 #ifdef HAVE_MPI
341     double start_time = MPI_Wtime();
342 #endif
343
344     nsd = nsd_;
345
346     preparePartition (nsd, global_num_cells, num_parts
          );
347     double* spacing = new double[3];
348     dx = spacing[0] = 1.0/global_num_cells[0];
349     dy = spacing[1] = 1.0/global_num_cells[1];
350     if (nsd >= 3) {
351       dz = spacing[2] = 1.0/global_num_cells[2];
```

```
352     }
353
354     const double d = 0.01;//Diffusion coeff. Beta =
            1/100;
355
356     dt = dt_;
357     theta = 0.5; // Crank-Nicholson
358     prepareMatrix (dt, d, theta, spacing, nsd,
            solver_type);
359
360     preparePreconditioner (solver_type);
361
362     const double pi = atan(1)*4;
363
364     int i,j,k,l;
365     double** y = (double**)malloc(num_local_pts*sizeof
            (double*));
366     for (k=0; k<num_local_pts; k++)
367         y[k] = (double*)malloc(9*sizeof(double));
368
369     // initial condition
370     k = 0;
371     if (nsd == 3 ) {
372         for (l=subd_lo_ix[2]; l<=subd_hi_ix[2]; l++) {
373             for (j=subd_lo_ix[1]; j<=subd_hi_ix[1]; j
                    ++) {
374                 for (i=subd_lo_ix[0]; i<=subd_hi_ix
                        [0]; i++) {
375                     y[k][0] = cos(2*pi*i*dx)*cos(2*pi*
                            j*dy)*cos(2*pi*l*dz);
376                     k++;
377                 }
378             }
379         }
380     }else if (nsd == 2) {
381         for (j=subd_lo_ix[1]; j<=subd_hi_ix[1]; j++) {
382             for (i=subd_lo_ix[0]; i<=subd_hi_ix[0]; i
                    ++) {
383                 y[k][0] = 0.0;
384                 y[k][1] = 0.0;
385                 y[k][2] = 0.0;
386                 y[k][3] = 0.0;
387                 y[k][4] = 1.0;
388                 y[k][5] = 1.0;
389                 y[k][6] = 0.0;
```

```
390                     y[k][7] = 1.0;
391                     y[k][8] = 0.0;
392                     if (i*dx>0.5)
393                         y[k][0] = 1.0;
394                     if (j*dy>0.5) {
395                         y[k][1] = 1.0;
396                     }
397                     ++k;
398                 }
399             }
400         }
401
402         //pycc::Panfilov_2order* system = new pycc::
                Panfilov_2order;
403         //system->v_rest = 0.0; system->v_peak = 1.0;
                // Panfilov specific
404
405         double t, ist;
406         double T = T_;  // Simulation time
407         double plot_dt = 100; //5;
408
409         double fm;
410
411         double tol = 1e-7;
412         int max_it = 500;
413         int test_conv = 1;
414
415         Jacobi jsolve;
416         if (strcmp(solver_type, "Jacobi")==0) {
417             jsolve.SetParameters(nsd, global_num_cells,
                        num_parts, subd_lo_ix,
418                         subd_hi_ix, subd_rank, global_indices,
                                num_local_pts, dt, delta);
419         }
420
421         for (t=dt; t<=T+0.1*dt; t += dt) {
422
423             fm = fmod(t,plot_dt);
424
425             // ODE half-step with 0.5*dt (using Strang
                    splitting)
426             for (k=0; k<num_local_pts; k++) {
427                 //system->forward(y[k],0.,0.5*dt);      //
                        Panfilov specific
428                 (*u_prev)[k] = y[k][0];
```

```
429             }
430
431             if (theta==0.0)
432                 (*rhs)[k] = scaling_vec[k]*(*u_prev)[k];
433             else {
434                 const double factor1 = 1.0/theta;
435                 const double factor2 = (1.0-theta)/theta;
436
437                 A->Multiply (false, *u_prev, *rhs);
438
439                 for (k=0; k<num_local_pts; k++)
440                     (*rhs)[k] = factor1*scaling_vec[k]*(*
                            u_prev)[k]-factor2*(*rhs)[k];
441             }
442
443             if (strcmp(solver_type, "CG")==0) {
444                 printf("<%03d> Running CG\n", my_id);
445                 linear_solver->Iterate (100, 1e-7);
446             }
447             if (strcmp(solver_type,"Jacobi")==0) {
448                 printf("<%03d> Running Jacobi\n", my_id);
449                 jsolve.SolveJacobi(A, rhs, u_prev, max_it,
                        tol, test_conv);
450
451                 for (k = 0; k<num_local_pts; k++) {
452                     (*u)[k] = jsolve.u_end[k];
453                 }
454             }
455
456
457             for (k=0; k<num_local_pts; k++) {
458                 y[k][0] = (*u)[k];
459             }
460             // ODE half-step with 0.5*dt (using Strang
                    splitting)
461             for (k=0; k<num_local_pts; k++) {
462                 //system->forward(y[k],0.,0.5*dt);      //
                        Panfilov specific
463                 (*u)[k] = y[k][0];
464             }
465
466             if (fm < 0.1*dt || fabs(fm-plot_dt) < 0.1*dt)
                    {  // save the solution
467                 printf("time to plot %f\n",t);
468                 double* vec;
```

```
469                 u->ExtractView(&vec);
470                 saveResults (vec, "E", t);
471             }
472         }
473 
474         for (k=0; k<num_local_pts; k++)
475             free (y[k]);
476         free (y);
477 
478         if (my_id==0)
479             if (nsd==3) {
480                 printf("Mesh=[%d x %d x %d], T=%g, dt=%g\n
                        ",
481                         global_num_cells[0],
                            global_num_cells[1],
                            global_num_cells[2], T, dt);
482             } else if (nsd==2) {
483                 printf("Mesh=[%d x %d], T=%g, dt=%g\n",
484                         global_num_cells[0],
                            global_num_cells[1], T, dt);
485             }
486 #ifdef HAVE_MPI
487         double stop_time = MPI_Wtime();
488         printf("<%03d> Total use of wall-clock time: %g\n"
                ,
489                 my_id,stop_time-start_time);
490 #endif
491 }
492 
493 void RectDomain::saveResults (double* vec, const char
        * v_name, double time)
494 {
495         int i,j, k=0, n;
496         const int nx = subd_hi_ix[0]-subd_lo_ix[0]+1;
497         const int ny = subd_hi_ix[1]-subd_lo_ix[1]+1;
498         const int nz = subd_hi_ix[2]-subd_lo_ix[2]+1;
499 
500         char filename[50];
501         sprintf(filename,"%s_T%07.1f_subd%05d",v_name,time
                ,my_id+1);
502         FILE* fp = fopen (filename,"wb");
503         fwrite (&nx, sizeof(int), 1, fp);
504         fwrite (&ny, sizeof(int), 1, fp);
505         if (nsd==3)
506             fwrite (&nz, sizeof(int), 1, fp);
```

```
507
508         double coord;
509         for (i=subd_lo_ix[0]; i<=subd_hi_ix[0]; i++) {
510             coord = i*dx;
511             fwrite (&coord, sizeof(double), 1, fp);
512         }
513
514         for (j=subd_lo_ix[1]; j<=subd_hi_ix[1]; j++) {
515             coord = j*dy;
516             fwrite (&coord, sizeof(double), 1, fp);
517         }
518         if (nsd >= 3) {
519             for (n=subd_lo_ix[2]; n<=subd_hi_ix[2]; n++) {
520                 coord = n*dz;
521                 fwrite (&coord, sizeof(double), 1, fp);
522             }
523         }
524         if (nsd == 2)
525             fwrite (vec, sizeof(double), nx*ny, fp);
526         else if (nsd == 3)
527             fwrite (vec, sizeof(double),nx*ny*nz, fp);
528         fclose (fp);
529 }
```

cpp/main.cpp

```
1  #include "Epetra_Map.h"
2  #include "Epetra_CrsMatrix.h"
3  #include "Epetra_Vector.h"
4  #include "Epetra_Version.h"
5  #include <malloc.h>
6
7  #include <RectDomain.h>
8  #ifdef HAVE_MPI
9  #include "mpi.h"
10 #endif
11
12 int main(int argc, char *argv[])
13 {
14 #ifdef HAVE_MPI
15   MPI_Init(&argc,&argv);
16 #endif
17
18   int my_id = 0;
19   int nsd = 3;
20   int* gnum_cells = (int*)malloc(nsd*sizeof(int));
```

```
21    int* parts = (int*)malloc(nsd*sizeof(int));
22    double dt = 0.1, T = 100;
23    double start_t, stop_t;
24    char *solver_type;
25    char s_type;
26
27  #ifdef HAVE_MPI
28    MPI_Comm_rank (MPI_COMM_WORLD, &my_id);
29  #else
30    parts[0] = parts[1] = parts[2] = 1;
31  #endif
32
33
34  #ifdef HAVE_MPI
35    if (my_id == 0) {
36    start_t = MPI_Wtime();
37    }
38  #endif
39
40    if (my_id==0) {
41      if (argc > 1) {
42        nsd = atof(argv[1]);
43      }
44
45      if (argc>2 && argv[2][0]=='[')
46        if (nsd == 3) {
47        sscanf(argv[2], "[%d,%d,%d]", &gnum_cells[0], &
              gnum_cells[1], &gnum_cells[2]);
48        } else if (nsd == 2) {
49          sscanf(argv[2], "[%d,%d]", &gnum_cells[0], &
                gnum_cells[1]);
50        }
51
52      else
53        if (nsd == 3)
54          gnum_cells[0] = gnum_cells[1] = gnum_cells[2]
              = 100;
55        else if (nsd == 2)
56          gnum_cells[0] = gnum_cells[1] = 100;
57
58
59      if (argc>3 && argv[3][0]=='[')
60        if (nsd == 3) {
61          sscanf(argv[3], "[%d,%d,%d]", &parts[0], &
                parts[1], &parts[2]);
```

```
62          } else if (nsd == 2) {
63            sscanf(argv[3], "[%d,%d]", &parts[0], &parts
                  [1]);
64          }
65
66        else
67          if (nsd == 3)
68            parts[0] = parts[1] = parts[2] = 1;
69          else if (nsd == 2)
70            parts[0] = parts[1] = 1;
71
72        if (argc>4) {
73          s_type = argv[4][0];
74          printf("Input: %c\n", s_type);
75        } else {
76          s_type = 'J';
77        }
78
79        if (argc>5)
80          dt = atof(argv[5]);
81
82        if (argc>6)
83          T = atof(argv[6]);
84
85        if (nsd == 3) {
86          printf("gnum_cells=[%d,%d,%d] parts=[%d,%d,%d]
                  dt=%g\n",
87         gnum_cells[0],gnum_cells[1],gnum_cells[2],parts
                  [0],parts[1],parts[2],      dt);
88        } else if (nsd == 2) {
89          printf("gnum_cells=[%d,%d] parts=[%d,%d] dt=%g\n
                  ",
90         gnum_cells[0],gnum_cells[1],parts[0],parts[1],dt)
                  ;
91        }
92
93      }
94
95    #ifdef HAVE_MPI
96      MPI_Bcast (&nsd, 1, MPI_INT, 0, MPI_COMM_WORLD);
97      MPI_Bcast (gnum_cells, nsd, MPI_INT, 0,
           MPI_COMM_WORLD);
98      MPI_Bcast (parts, nsd, MPI_INT, 0, MPI_COMM_WORLD);
99      MPI_Bcast (&dt, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
100     MPI_Bcast (&T, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
101   MPI_Bcast (&s_type, 1, MPI_CHAR, 0, MPI_COMM_WORLD);
102   if (s_type == 'J' || s_type == 'j') {
103     solver_type = "Jacobi";
104   } else if (s_type == 'C' || s_type == 'c') {
105     solver_type = "CG";
106   }
107   printf("<%03d> Solver type: %s\n", my_id,
        solver_type);
108 #endif
109   RectDomain problem;
110   problem.work (dt, T, gnum_cells, parts, solver_type,
        nsd);
111
112 #ifdef HAVE_MPI
113   if (my_id == 0) {
114     stop_t = MPI_Wtime();
115     printf("Total time used as calculated by process
          0: %g\n",stop_t - start_t);  }
116   MPI_Finalize ();
117 #endif
118
119   return 0;
120 }
```

cpp/PlotRes.py

```
1  import numpy as np
2  #from struct import *
3  import struct
4  import matplotlib.pyplot as plt
5  from mpl_toolkits.mplot3d import Axes3D
6  from math import pi, exp, cos
7
8  Lx = 1.
9  Ly = 1.
10 Lz = 1.
11 h = 0.5**7
12 dx = h
13 dy = h
14 dz = h
15
16 nsd = 3
17 mpiprocs = 64; #Gives the number of files to be read.
     Has to be set manually
18
19 if nsd == 3:
```

```
20      N = [Lx/dx+1,Ly/dy+1,Lz/dz+1]
21      Nx = N[0]
22      Ny = N[1]
23      Nz = N[2]
24  elif nsd == 2:
25      N = [Lx/dx,Ly/dy]
26      Nx = N[0]
27      Ny = N[1]
28
29  print N
30  u = np.zeros(Nx*Ny*Nz)
31
32  def meshgrid2(*arrs):
33    #Meshgrid for more than 2D
34      arrs = tuple(reversed(arrs))
35      lens = map(len, arrs)
36      dim = len(arrs)
37
38      sz = 1
39      for s in lens:
40          sz*=s
41
42      ans = []
43      for i, arr in enumerate(arrs):
44          slc = [1]*dim
45          slc[i] = lens[i]
46          arr2 = np.asarray(arr).reshape(slc)
47          for j, sz in enumerate(lens):
48              if j!=i:
49                  arr2 = arr2.repeat(sz, axis=j)
50          ans.append(arr2)
51
52      return tuple(ans)
53
54  #Reading the binary result files
55  for p in range(1,mpiprocs+1):
56    if p < 10:
57          fileName = "E_T00005.0_subd0000{0}".format(p)
58    elif p < 100:
59          fileName = "E_T00005.0_subd000{0}".format(p)
60    elif p < 1000:
61      fileName = "E_T00005.0_subd00{0}".format(p)
62    else:
63      fileName = "E_T00005.0_subd0{0}".format(p)
64
```

```
65      with open(fileName, mode='rb') as file:
66          fileContent = file.read()
67
68      print "Reading file"
69
70      nx = struct.unpack("i",fileContent[:4])
71      nx = reduce(lambda rst, d: rst * 10 + d, nx)
72      ny = struct.unpack("i",fileContent[4:8])
73      ny = reduce(lambda rst, d: rst * 10 + d, ny)
74      if nsd == 3:
75          nz = struct.unpack("i",fileContent[8:12])
76          nz = reduce(lambda rst, d: rst * 10 + d, nz)
77      if nsd == 2:
78          icoord = struct.unpack("d"*nx,fileContent[8:8*
                nx+8])
79          jcoord = struct.unpack("d"*ny,fileContent[8*nx
                +8:8*(ny+nx)+8])
80          u_part = struct.unpack("d"*(nx*ny),fileContent
                [8*(ny+nx)+8:])
81
82      elif nsd == 3:
83          icoord = struct.unpack("d"*nx,fileContent
                [12:8*nx+12])
84          jcoord = struct.unpack("d"*ny,fileContent[8*nx
                +12:8*(ny+nx)+12])
85          kcoord = struct.unpack("d"*nz,fileContent[8*(
                nx+ny)+12:8*(nx+ny+nz)+12])
86          u_part = struct.unpack("d"*(nx*ny*nz),
                fileContent[8*(ny+nx+nz)+12:])
87
88      I0 = int(round(icoord[0]/dx))
89      J0 = int(round(jcoord[0]/dy))
90      K0 = int(round(kcoord[0]/dz))
91
92      ic = 0
93      if nsd == 3:
94          for k in range(0,nz):
95              for j in range(0,ny):
96                  for i in range(0,nx):
97                      ic_glob = (K0+k)*(Nx*Ny)+(J0+j)*Nx
                            +(I0+i)
98                      u[ic_glob] = u_part[ic]
99                      ic += 1
100     elif nsd == 2:
101         for j in range(0,ny):
```

```
102                 for i in range(0,nx):
103                     ic_glob = (J0+j)*Nx+(I0+i)
104                     u[ic_glob] = u_part[ic]
105                     ic += 1
106 
107 
108 
109 beta = 100
110 T_end = 5.
111 x = np.linspace(0,Lx,Nx)
112 y = np.linspace(0,Ly,Ny)
113 z = np.linspace(0,Lz,Nz)
114 [X,Y,Z] = meshgrid2(x,y,z)
115 v0 = np.cos(2*pi*X)*np.cos(2*pi*Y)*np.cos(2*pi*Z)
116 u_an = exp((-3*(2*pi)**2/beta)*T_end)*v0
117 u_an_plot = np.reshape(u_an,(Nx,Ny,Nz))
118 [X,Y] = np.meshgrid(x,y)
119 fig = plt.figure()
120 ax = fig.add_subplot(111,projection='3d')
121 u_plot = np.reshape(u,(Nx,Ny,Nz))
122 ax.plot_surface(X,Y,u_plot[:,:,1])
123 fig2 = plt.figure()
124 ax2 = fig2.add_subplot(111,projection='3d')
125 ax2.plot_surface(X,Y,u_an_plot[:,:,1])
126 err = u_plot-u_an_plot
127 fig3 = plt.figure()
128 ax3 = fig3.add_subplot(111,projection='3d')
129 ax3.plot_surface(X,Y,err[:,:,1])
130 plt.show()
```

Bibliography

Chai, Jun et al. (2013). "Towards simulation of subcellular calcium dynamics at nanometre resolution". In: *International Journal of High Performance Computing Applications*, p. 1094342013514465.

charm.cs.illinois.edu (2015). *2DJacobi_ NeighborComm.jpg*. URL: `https://charm.cs.illinois.edu/tutorial/images/2DJacobi_NeighborComm.jpg`.

Eide, Vegard and Einar Næss Jensen (2015). *About Vilje*. URL: `https://www.hpc.ntnu.no/display/hpc/About+Vilje` (visited on 09/15/2015).

Hackbusch, Wolfgang (1994). *Iterative Solution of Large Sparse Systems of Equations*. Springer-Verlag.

Hanslien, Monica et al. (2011). "Stability of two time-integrators for the alievpanfilov system". In: *International Journal of Numerical Analysis and Modeling* 8.3, pp. 427–442.

Langtangen, Hans Petter (2014). *Truncation Error Analysis*. URL: `http://hplgit.github.io/num-methods-for-PDEs/doc/pub/trunc/pdf/trunc-4screen.pdf` (visited on 04/07/2015).

Lyche, Tom (2013). *Lecture Notes for MAT-INF4130*. URL: `http://www.uio.no/studier/emner/matnat/math/MAT-INF4130/h14/book2013.pdf` (visited on 08/19/2014).

Mardal, Kent-Andre and Anders Logg (2013). *Lectures on the Finite Element Method*. URL: `http://folk.uio.no/kent-and/book.pdf` (visited on 08/19/2014).

Saad, Yousef (2003). *Iterative methods for Sparse Linear Systems*. 2nd ed.

Shönlieb, C.-B. (2013). *Numerical Analysis - Lecture 17*. URL: `http://www.damtp.cam.ac.uk/user/cbs31/Teaching_files/c17.pdf` (visited on 09/20/2015).

Trilinos (2015). *Trilinos Home Page*. URL: `https://trilinos.org/` (visited on 09/15/2015).

Tveito, Aslak and Ragnar Winther (2009). *Introduction to Partial Differential Equations. A computational Approach*. Springer.

www.prace-ri.eu (2015). *experiments.jacobi.fig_jacobi_ 3d_ comm-c3539.png*. URL: `http://www.prace-ri.eu/local/cache-vignettes/L100xH0/experiments.jacobi.fig_jacobi_3d_comm-c3539.png`.

Yang, Xiyang IA and Rajat Mittal (2014). "Acceleration of the Jacobi iterative method by factors exceeding 100 using scheduled relaxation". In: *Journal of Computational Physics* 274, pp. 695–708.